1. 1.0 Introducción a la Edición de Connexions
2. Introducción al Cómputo de Alto Rendimiento
3. Arquitecturas de Cómputo Modernas
    1. Memoria
        1. Introducción
        2. Tecnologías de Memoria
        3. Registros
        4. Caches
        5. Organización de la Cache
        6. Memoria Virtual
        7. Mejorando el Rendimiento de la Memoria
        8. Notas de Cierre
        9. Ejercicios
    2. Números de Punto Flotante
        1. Introduction
        2. Reality
        3. Representation
        4. Effects of Floating-Point Representation
        5. More Algebra That Doesn't Work
        6. Improving Accuracy Using Guard Digits
        7. History of IEEE Floating-Point Format
        8. IEEE Operations
        9. Special Values
        10. Exceptions and Traps
        11. Compiler Issues
        12. Closing Notes
        13. Exercises
4. Programando y Afinando Software
    1. Qué Hace un Compilador
        1. Qué Hace un Compilador - Introducción

    2. Qué Hace un Compilador - Historia de los Compiladores
    3. Qué Hace un Compilador - Cuál Lenguaje Optimizar
    4. Qué Hace un Compilador - Visita Guiada al Optimizador de Código
    5. Qué Hace un Compilador - Niveles de Optimización
    6. Qué Hace un Compilador - Optimizaciones Clásicas
    7. Qué Hace un Compilador - Notas de Cierre
    8. Qué Hace un Compilador - Ejercicios
2. Cronometraje y Perfilado
    1. Cronometraje y Perfilado - Introducción
    2. Cronometraje y Perfilado - Cronometraje
    3. Cronometraje y Perfilado - Perfilado de Subrutinas
    4. Cronometraje y Perfilado - Perfiladores de Bloques Básicos
    5. Cronometraje y Perfilado - Memoria Virtual
    6. Cronometraje y Perfilado - Notas de Cierre
    7. Cronometraje y Perfilado - Ejercicios
3. Eliminando el Desorden
    1. Eliminando el Desorden - Introducción
    2. Eliminando el Desorden - Llamado a Subrutinas
    3. Eliminando el Desorden - Bifurcaciones
    4. Eliminando el Desorden - Bifurcaciones con Ciclos
    5. Eliminando el Desorden - Otras Formas de Desorden
    6. Eliminando el Desorden - Notas de Cierre
    7. Eliminando el Desorden - Ejercicios
4. Optimización de Ciclos
    1. Optimización de Ciclos - Introducción
    2. Optimización de Ciclos - Conteo de Operaciones
    3. Optimización de Ciclos - Desenrollado Básico de Ciclos

    4. Optimización de Ciclos - Calificando a los Candidatos para el Desenrollado de Ciclos
    5. Nested Loops
    6. Loop Interchange
    7. Memory Access Patterns
    8. When Interchange Won't Work
    9. Blocking to Ease Memory Access Patterns
    10. Programs That Require More Memory Than You Have
    11. Closing Notes
    12. Exercises
5. Procesadores Paralelos de Memoria Compartida
    1. Entendiendo el Paralelismo
        1. Entendiendo el Paralelismo - Introducción
        2. Entendiendo el Paralelismo - Dependencias
        3. Loops
        4. Loop-Carried Dependencies
        5. Ambiguous References
        6. Closing Notes
        7. Exercises
    2. Shared-Memory Multiprocessors
        1. Introduction
        2. Symmetric Multiprocessing Hardware
        3. Multiprocessor Software Concepts
        4. Techniques for Multithreaded Programs
        5. A Real Example
        6. Closing Notes
        7. Exercises
    3. Programming Shared-Memory Multiprocessors
        1. Introduction
        2. Automatic Parallelization
        3. Assisting the Compiler

        4. Closing Notes
        5. Exercises
6. Procesamiento Paralelo Escalable
    1. Soporte del Lenguaje para Mejorar el Rendimiento
        1. Soporte del Lenguaje para Mejorar el Rendimiento - Introducción
        2. Soporte del Lenguaje para Mejorar el Rendimiento - Problema de Datos en Paralelo: Flujo Calórico
        3. Soporte del Lenguaje para Mejorar el Rendimiento - Lenguajes Explícitamente Paralelos
        4. Soporte del Lenguaje para Mejorar el Rendimiento - FORTRAN 90
        5. Soporte del Lenguaje para Mejorar el Rendimiento - Descomposición del Problema
        6. Soporte del Lenguaje para Mejorar el Rendimiento - FORTRAN de Alto Rendimiento (HPF)
        7. Soporte del Lenguaje para Mejorar el Rendimiento - Notas Finales
    2. Ambientes de Paso de Mensajes
        1. Ambientes de Paso de Mensajes - Introducción
        2. Ambientes de Paso de Mensajes - Parallel Virtual Machine
        3. Message-Passing Interface
        4. Closing Notes
7. Apéndices
    1. Appendix C: High Performance Microprocessors
        1. Introduction
        2. Why CISC?
        3. Fundamental of RISC
        4. Second-Generation RISC Processors
        5. RISC Means Fast
        6. Out-of-Order Execution: The Post-RISC Architecture

    7. Closing Notes
    8. Exercises
2. Appendix B: Looking at Assembly Language
    1. Assembly Language

1.0 Introducción a la Edición de Connexions
Proporciona una introducción a la versión del libro re-publicada en Connexions.

Introducción a la Edición de Connexions

El propósito de este libro siempre ha sido enseñar a los nuevos programadores y científicos las bases del Cómputo de Alto Rendimiento (HPC, High Performance Computing, por sus siglas en inglés). Hay muchos libros sobre paralelismo y cómputo de alto rendimiento que se enfocan en los aspectos de la ciencia computacional, la teoría y la arquitectura que rodean al HPC. Yo quiero que este libro vaya dirigido al estudiante práctico de química, física o biología que necesita escribir y ejecutar sus programas como parte de su trabajo de investigación. Yo usaba la primera edición del libro escrito por Kevin Dowd en 1996, hasta que descubrí que ya no se reimprimía. Inmediatamente envié una carta a O'Reilly quejándome por ello, e implorándoles que lo mantuviesen en circulación, pues es el único de su tipo en el mercado. Dicha carta desencadenó varias conversaciones que terminaron por convertirme en el autor de la segunda edición. En un estilo completamente "open source" -con el que estoy de acuerdo-, encontré y resolví la falla. Durante el otoño de 1997, mientras usaba el libro como texto de mi curso de HPC, lo fui reescribiendo un capítulo a la vez, impulsado por múltiples tazas nocturnas de café latte y el miedo de no tener nada listo para mi clase de esa semana.

La segunda edición apareció en julio de 1998, y tuvo una gran recepción. Obtuve muchos buenos comentarios de profesores y científicos, quienes sentían que el libro hacía un buen trabajo al enseñar al practicante, lo cuál me hace muy feliz.

En 1998, este libro se publicó en un punto de inflexión en la historia del Cómputo de Alto Rendimiento. A fines de la década de 1990 aún quedaba la pregunta de si las grandes supercomputadoras vectoriales, con sus sistemas especializados e memoria, podían resistir el ataque de las crecientes velocidades de reloj de los microprocesadores. También a fines de esa década nos preguntábamos si las rápidas, caras y energéticamente voraces arquitecturas RISC podía ganar a los microprocesadores

comerciales de Intel, y a las tecnologías de memoria comercialmente disponibles.

Para el 2003, el mercado había dado la victoria al microprocesador -su rendimiento y el de los subsistemas de memoria seguían incrementándose rápidamente. Para el 2006, la arquitectura Intel ha eliminado a todos los procesadores de arquitectura RISC, al incrementar grandemente sus velocidades de reloj y ganar otra competencia muy importante, la de las Operaciones de Punto Flotante por Watt consumido. Una vez que los usuarios imaginaron cómo usar de forma efectiva los procesadores débilmente acoplados, el costo global y el consumo de energía mejorados de los procesadores comerciales se convirtieron en los factores primordiales del mercado.

Tales cambios hicieron que el libro se volviera cada vez menos relevante para los casos de uso común en el campo del HPC, y condujeron a su no republicación -para gran disgusto de su pequeña pero fiel base de fanáticos. Se redujo a comprar en Amazon copias usadas del libro con el objeto de tener unas pocas copias circulando por la oficina, para dárselas como regalos a visitantes que no se lo esperaban.

Gracias al enfoque visionario de O'Reilly y Asociados de usar los derechos reservados de los Fundadores, y liberar los libros descontinuados bajo la Atribución Creative Commons, este libro pudo resurgir una vez más de sus cenizas como el proverbial Fénix. Al dar este libro a Connexions y publicarlo bajo una licencia de Atribución Creative Commons estamos asegurando que el libro jamás quedará nuevamente obsoleto. Podemos tomar los elementos clave del libro que todavía son relevantes, y una nueva comunidad de autores pueden añadir y adaptar el libro conforme se necesite en tiempos venideros.

Publicarlo a través de Connexions también asegura que el costo de los libros impresos sea muy bajo, convirtiéndolo en una buena decisión como libro de texto para cursos universitarios de Cómputo de Alto Rendimiento. El Licenciamiento Creative Commons y la habilidad de imprimirlo localmente, hace que el libro esté disponible en cualquier país y escuela del Mundo. Como Wikipedia, aquellos de nosotros que usan el libro pueden

convertirse en los voluntarios que ayudarán a mejorarlo, a la vez que se convierten en coautores del mismo.

Debo dar las gracias a Kevin Dowd, quien escribió la primera edición y amablemente me permitió alterarla de portada a contraportada en la segunda edición. Mike Loukides de O'Reilly fue el editor de ambas ediciones, y hablamos de vez en cuando sobre una posible edición futura del libro. Mike también fue una ayuda fundamental para liberar el libro de O'Reilly bajo la Atribución Creative Commons. Ha sido maravilloso trabajar con el equipo en Connexions. Compartimos una pasión por el Cómputo de Alto Rendimiento y las nuevas formas de publicación, de forma que el conocimiento alcance a tanta gente como sea posible. Quiero agradecer a Jan Odegard y Kathi Fletcher por darme el valor, apoyarme y ayudarme a través de todo el proceso de re-publicación. Daniel Williamson hizo un trabajo sorprendente al convertir el material de los formatos de O'Reilly a los de Connexions.

Realmente espero ver qué tan adelante irá ahora este libro, de forma que podamos tener un número ilimitado de coautores que inviertan su tiempo y lo utilicen. Espero con interés poder trabajar con todos ustedes.

Charles Severance - 12 de noviembre de 2009.

Introducción al Cómputo de Alto Rendimiento
Esta es una introducción al libro Cómputo de Alto Rendimiento.

¿Por qué Preocuparse por el Rendimiento?

Durante la última década, la definición de lo que se denomina "cómputo de alto rendimiento" ha cambiado dramáticamente. En 1988 apareció un artículo en el Wall Street Journal titulado "El Ataque de los Micros Asesinos", que describía la forma en que los sistemas de cómputo compuestos de varios pequeños procesadores económicos pronto haría obsoletas a las grandes supercomputadoras. En ese tiempo, una "computadora personal" que costaba US$3,000 podía realizar 0.25 millones de operaciones de punto flotante por segundo, una "estación de trabajo" que costaba US$20,000 podía realizar 3 millones de operaciones de punto flotante, y una supercomputadora que costaba US$3 millones podía realizar 100 millones de operaciones de punto flotante por segundo. Así que, ¿por qué no simplemente conectar 400 computadoras personales juntas para lograr el mismo rendimiento de una supercomputadora, por tan sólo US$1.2 millones?

Esta visión se ha hecho realidad en varias formas, pero no en la que pensaban aquellos que propusieron originalmente la idea de los "micros asesinos". En vez de ello, el rendimiento del microprocesador ha ganado implacablemente al de la supercomputadora. Esto ha sucedido por dos razones. Primero, se ha dedicado mucha más inteligencia a mejorar el rendimiento en el área de la computadora personal, que la aplicada a las supercomputadoras de los ochenta para el mismo fin. Además, una vez qu las compañías de supercómputo logran romper alguna barrera técnica, las compañías de microprocesadores pueden adoptar rápidamente tales elementos exitosos de los diseños de supercómputo, con pocos años de diferencia. El segundo factor, tal vez el más importante, fue la emergencia de un próspero mercado de la computadora personal y de negocios, con demandas cada vez mayores de rendimiento. Usos computacionales tales como las gráficas 3D, interfaces gráficas de usuario, multimedia y videojuegos fueron los factores impulsores de este mercado. Con un mercado tan grande, fluyero los dólares disponibles para la investigación y el desarrollo de procesadores económicos de alto rendimiento para el

mercado casero. El resultado de esta tendencia hacia computadoras más rápidas y pequeñas, se hace evidente conforme los antiguos fabricantes de supercomputadoras están siendo adquiridos por compañías que fabrican estaciones de trabajo (Silicon Graphics compró Cray, y Hewlett-Packard compró Convex en 1996).

Como resultado, casi todo aquel con acceso a una computadora tiene ahora un procesador de "alto rendimiento". Conforme crecen las velocidades pico de esas nuevas computadoras personales, estas máquinas encuentra todos los retos de rendimiento típicos de las supercomputadoras.

Aunque no todos los usuarios de estaciones de trabajo personales requieran conocer los detalles íntimos del cómputo de alto rendimiento, aquellos quienes programan estos sistemas para extraerles el máximo rendimiento se beneficiarán de un entendimiento de las fortalezas y debilidades de estos novedosos sistemas de alto desempeño.

Alcances del Cómputo de Alto Rendimiento

El cómputo de alto rendimiento cubre un amplio espectro de sistemas, desde nuestras computadoras de escritorio hasta los grandes sistemas de procesamiento paralelo. Dado que la mayoría de los sistemas de alto rendimiento están basados en procesadores para *computadoras con conjunto reducido de instrucciones* (RISC por sus siglas en inglés), muchas técnicas aprendidas en cierto tipo particular de sistemas se transfieren fácilmente a los otros.

Los procesadores RISC de alto rendimiento están diseñados de forma tal que se puedan insertar fácilmente en un sistema multiprocesador, de entre 2 y 64 CPUs accesando a una memoria única, usando *multiprocesamiento simétrico* (SMP por sus siglas en inglés). Pero programar múltiples procesadores para resolver un único problema genera su propio conjunto de retos adicionales para el programador, quien debe estar consciente de cuántos de esos procesadores múltiples operan juntos, y cómo puede dividirse eficientemente el trabajo entre ellos.

Incluso en aquellos casos done cada procesador es muy poderoso, y puede ponerse un pequeño número de ellos en un único contenedor, a menudo existirán aplicaciones tan grandes que requieran distribuirse en varios contenedores. Para poder cooperar en la resolución de una aplicación más grande, estos contenedores se enlazan entre sí mediante una red de alta velocidad, de modo que operen como una *red de estaciones de trabajo* (NOW por sus siglas en inglés). Puede usarse una NOW individualmente como un sistema de encolamiento por lotes, o como una multicomputadora más grande, empleando una herramienta de paso de mensajes tal como la *máquina virtual paralela* (PVM por sus siglas en inglés) o la *interfaz de paso de mensajes* (MPI).

Para los problemas más granes, aquellos con más interacción entre datos y cuyos usuarios manejan presupuestos del orden de millones de dólares, todavía existe el extremo superior del espectro de computadoras de alto rendimiento, los sistemas de procesamiento paralelo escalable con centenares a millares de procesadores. Tales sistemas vienen en dos variedades. Una de ellas es programable usando paso de mensajes. En vez de usar una red de área local estándar, tales sistemas se conectan usando una interconexión propietaria, escalable, de gran ancho de banda y baja latencia (¿a poco no parece charla de mercadólogo?). Gracias a esta interconexión de alto rendimiento, esto istemas pueen escalar hasta miles de procesadores, a la vez que minimizan el tiempo utilizado (gastado) en la sobrecarga debida a las comunicaciones en sí.

El segundo tipo de sistema de procesamiento paralelo es el denominado *acceso a memoria no-uniforme escalable* (NUMA por sus siglas en inglés). Tales sistemas también usan un mecanismo de interconexión de alto rendimiento entre los procesadores, pero en vez de usarlo para intercambiar mensajes, lo emplean para instrumentar una memoria compartida distribuida, accesible para cualquier procesador a través del paradigma carga/almacenamiento. En este sentido es similar a programar sistemas SMP, excepto que el acceso a algunas zonas de memoria es más lento que a otras.

Estudiando Cómputo de Alto Rendimiento

Estudiar cómputo de alto rendimiento es una excelente excusa para repasar lo que sabemos de arquitectura de computadoras. Una vez en pos de extraer hasta el último bit de rendimiento de nuestros sistemas de cómputo, estaremos más motivados para comprender plenamente aquellos aspectos de la arquitectura que tienen un impacto directo en el rendiminto del sistema.

A lo largo de toda la historia de la computación, los vendedores nos han dicho que sus compiladores resolverán todos nuestros problemas, y que los creadores de tales compiadores pueden lograr el mejor rendimiento absoluto del hardware subyacente. Tal reclamo nunca ha sido, y probablemente nunca será, totalmente cierto. La habilidad del compilador para lograr el rendimiento máximo disponible en el hardware mejora con cada nueva generación de ambos, hardware y software. Sin embargo, conforme ascendemos en la jerarquìa de las arquitecturas de cómputo de alto rendimiento, podemos depender cada vez menos del compilador, y los programadores deben tomar a su cargo la responsabilidad del rendimiento de su código.

En los sistemas con un solo procesador y en los SMP con pocas CPUs, uno de nuestros objetivos como programadores debe ser quitarnos del camino y no estorbar al compilador. Con frecuencia los constructos usados para mejorar el rendimiento en una arquitectura en particular, limitan nuestra habilidad de mejorar el rendimiento en otra aquitectura. Es más, tales "brillantes" (léase obtusas) optimizaciones manuales a menudo confunden al compilador, limitando su habilidad de transformar automáticamente nuestro código para que tome ventaja de las fortalezas particulares de la arquitectura subyacente.

Como programadores, es importante que conozcamos cómo trabaja el compilador, de forma que podamos saber cuándo ayudarlo, y cuándo hacernos a un lado. También debemos estar conscientes que conforme mejoran los compiladores (aunque nunca sea tanto como dicen los vendedores), es mejor delegar más responsabilidad en ellos.

Conforme ascendemos en la jerarquía de las computadoras de alto rendimiento, necesitamos aprender nuevas técnicas para mapear nuestros programas hacia esas arquitecturas, incluyendo extensiones de lenguajes,

llamadas a bibliotecas y directivas de compilación. Conforme usamos estas características, nuestros programas se vuelven menos transportables. También, al utilizar estos constructos de alto nivel, no deberemos hacer modificaciones que resulten en un rendomiento pobre sobre los microprocesadores RISC individuales que a menudo conforman el sistema de procesamiento paralelo.

Midiendo el Rendimiento

Cuando se adquiere una computadora para aplicaciones computacionalmente intensivas, es importante determinar qué tan bien desempeñará esta función el sistema. Una forma de elegir uno de entre un conjunto de sistemas contendientes, es que cada vendedor le preste uno de sus sistemas durante un periodo de tiempo, para probar las aplicaciones. Al final de tal periodo de evaluación, puede usted devolver aquellos que no dieron el ancho y pagar por su favorito. Desafortunadamente, muchos vendedores no le prestan sus equipos durante tal periodo de tiempo a menos que haya alguna seguridad de que eventualmente lo comprará.

Con frecuencia evaluaremos el rendimiento potencial del sistema usando *benchmarks*. Hay benchmarks industriales, así como los suyos propios. Ambos tipos requieren de pensamiento y planeación cuidadosos, si se quiere que sean una herramienta efectiva para determinar el mejor sistema para su aplicación.

El Siguiente Paso

Independientemente del aspecto económico, el rendimiento computacional es un tema fascinante y retador. La arquitectura de cómputo es interesante por derecho propio, un tópico con el que todo profesional de la computación debe sentirse cómodo. Obtener hasta el último bit de rendimiento de una aplicación importante, puede ser un ejercicio estimulante, además de una necesidad económica. Probablemente hay algunas personas que simplemente disfrutan de unir ingenio con una arquitectura de cómputo inteligente.

¿Qué necesita para entrar al juego?

- Una comprensión básica de las arquitecturas de cómputo modernas. No necesita un grado avanzado en ingeniería de cómputo, pero sí cuando menos entender la terminología básica.
- Una comprensión básica de cómo realizar un benchmark, o medida de rendimiento, de forma que pueda cuantificar sus propios éxitos y fracasos, y usar esa información para mejorar el rendimiento de su aplicación.

Este libro pretende ser una introducción fácil de entender, así como una panorámica del cómputo de alto rendimiento. Es un campo interesante, y uno que se hará más importante conforme demandemos aún más a nuestras computadoras personales comunes. En el campo del cómputo de alto rendimiento, siempre hay una solución de compromiso entre el rendimiento de una sola CPU y el de un sistema con múltiples procesadores. Estos últimos generalmente son más caros y difíciles de programar (a menos que tenga usted este libro).

Algunas personas aseguran que eventualmente tendremos CPUs individuales tan rápidas, que no necesitaremos ningún tipo de arquitectura avanzada que requiera de habilidades especiales para programarla.

Hasta ahora en este campo de la informática, incluso a pesar de que el rendimiento del microprocesador económico ha incrementado en mil veces, no parece disminuir el interés en atar mil de esos procesadores, para incrementar la potencia en un millón de veces. Conforme más barato se vuelva el bloque constitutivo del cómputo de alto rendimiento, mayor será el beneficio de usar muchos procesadores. Si en algún momento en el futuro tenemos un solo procesador que sea más rápido que cualquiera de los sistemas escalables de 512 procesadores de hoy, piense cuánto podremos hacer cuando conectemos 512 de esos nuevos procesadores para formar un nuevo sistema.

De eso se trata este libro. Si le interesa, continúe leyendo.

Memoria

Supongamos que cierta noche se durmió temprano y comenzó a soñar. En su sueño, tiene una máquina del tiempo y unos pocos procesadores superescalares de 4 vías a 500 MHz. Programa su máquina del tiempo para regresar a 1981, y una vez en esa época, sale y compra una IBM PC con un microprocesador Intel 8088 corriendo a 4.77 MHz. Durante buena parte del resto de esa noche, da vueltas en la cama mientras trata de adaptar el procesador a 500 MHz al zócalo del Intel 8088, usando un cautín y una navaja suiza. Justo antes de despertar, la nueva computadora finalmente funciona, y la enciende para ejecutar el benchmark Linpack[footnote] y emite un comunicado de prensa. ¿Cabe esperar que esto convierta el sueño en una pesadilla? Existe una buena posibilidad de que suceda, tal como si la noche anterior hubiese usted regresado a la Edad Media y puesto un motor a reacción a un caballo. (debe dejar de comer pizzas con doble pepperoni tan tarde en la noche).
Véase [link]Capítulo 15, Usando Benchmarks Publicados, para detalles acerca del benchmark Linpack.

Incluso aunque pudiera acelerar los aspectos computacionales de un procesador infinitamente rápido, deberá cargar y almacenar los datos y las instrucciones desde y hacia una memoria, respectivamente. Los procesadores modernos continúan apegándose muy de cerca a este proceso infinitamente rápido. Pero el rendimiento de la memoria incrementa a una tasa mucho menor (le tomará más tiempo a la memoria volverse infinitamente rápida). Muchos de los problemas interesantes en el cómputo de alto rendimiento utilizan una gran cantidad de memoria. Conforme las computadoras se vuelven más rápidas, el tamaño de los problemas con los que tienden a operar también crece. El problema es que cuando quiere usted resolver esos problemas a altas velocidades, necesita un sistema de memoria que es grande, a la vez que rápido -un gran reto. Algunos enfoques posibles son los siguientes:

- Cada componente del sistema de memoria puede hacerse lo suficientemente rápido, de manera individual, para responder a cada

solicitud de acceso a memoria.
- Puede accederse a la memoria lenta en un estilo round-robin (con suerte), para lograr un efecto similar al de un sistema de memoria más rápido.
- Puede "ensancharse" el diseño del sistema de memoria, de modo que cada transferencia contenga muchos bytes de información.
- El sistema puede dividirse en porciones más rápidas y más lentas, y acomodarlas de forma que las primeras se usen más a menudo que las últimas.

De nuevo, la economía es la fuerza dominante en el negocio de las computadoras. Un sistema de memoria barato, optimizado estadísticamente se venderá mucho mejor que uno brillantemente rápido y prohibitivamente caro, de forma que la primera opción no es en realidad tal cosa. Pero estas opciones, usadas en combinación, pueden lograr una buena fracción del rendimiento que obtendría usted si cada componente fuera rápido. Hay muy buenas posibilidades de que su estación de trabajo de alto rendimiento incorpore varias o todas de estas opciones.

Una vez decidido el sistema de memoria, hay cosas que podemos hacer mediante software para ver que se use eficientemente. Un compilador que posee cierto conocimiento sobre la distribución de la memoria y los detalles del cache, puede usarlo para optimizar su uso hasta cierto punto. El otro punto propenso a optimizarse son las aplicaciones de usuario, como veremos más adelante en este libro. Un buen patrón de acceso a memoria trabajará con, y no en contra de, los componentes del sistema.

En este capítulo discutiremos cómo trabajan las piezas de un sistema de memoria. Veremos cómo los patrones de acceso a datos e instrucciones son relevantes en el tiempo de ejecución global, especialmente conforme incrementa la velocidad de la CPU. También hablaremos un poco acerca de las implicaciones que tiene para el rendimiento, el ejecutar los programas en un ambiente de memoria virtual.

Tecnologías de Memoria

Prácticamente todas las memorias rápidas actuales están basadas en semiconductores. [footnote] Vienen en dos variedades: *memoria dinámica de acceso aleatorio* (DRAM) y *memoria estática de acceso aleatorio* (SRAM). El término *aleatorio* significa que puede usted acceder a las localidades de memoria en cualquier orden. Se usa para distinguir el acceso aleatorio de las memorias seriales, en las que debe recorrer paso a paso todas las celdas intermedias hasta llegar a aquella en particular que le interesa. Un ejemplo de un medio de almacenamiento *no* aleatorio es la cinta magnética. Los términos dinámico y estático tienen que ver con la tecnología usada en el diseño de las celdas de memoria. Los dispositivos DRAM se basan en carga eléctrica, pues cada bit se representa mediante la carga almacenada por un diminuto capacitor. Dicha carga se fuga en un corto periodo de tiempo, asì que el sistema debe refrescarla continuamente para evitar la pérdida de datos. También el acto de leer un bit de la DRAM la descarga, y debe refrescarse. Y no es posible leer un bit de memoria DRAM mientras se está refrescando.
todavía se usa la memoria de núcleo magnético en aplicaciones donde la "dureza" de la radiación -resistencia a cambios causada por radiación ionizante- es importante.

La SRAM se basa en compuertas, y cada bit se almacena mediante un arreglo de cuatro a seis transistores conectados. Las memorias SRAM retienen sus datos mientras tengan energía, sin la necesidad de ningún mecanismo de refresco.

La DRAM ofrece la mejor tasa precio/rendimiento, así como la mayor densidad de celdas de memoria por chip. Ello significa un menor costo, menos espacio en las tarjetas, menos gasto energético y menos calor. Por otra parte, algunas aplicaciones, tales como la cache y la memoria de video, requieren velocidades mayores, para las que la SRAM resulta más adecuada. Actualmente, puede usted elegir entre SRAM y DRAM a velocidades por debajo de los 50 nanosegundos (ns). La SRAM tiene tiempos de acceso de alrededor de 7 ns a costa de mayor costo, calor, energía y espacio en la tarjeta.

El rendimiento de la memoria está limitado, además de por la tecnología básica necesaria para almacenar los bits de datos, por consideraciones prácticas tales como el acomodo de los alambres en el circuito integrado y las patillas externas para comunicar la información sobre direcciones y datos entre la memoria y el procesador.

Tiempos de Acceso

La cantidad de tiempo que toma leer o escribir una posición de memoria se denomina el *tiempo de acceso a memoria*. Una cantidad relacionada es el *tiempo de ciclo de memoria*. Mientras que el tiempo de acceso nos dice cuan rápidamente puede referenciar una posición de memoria, el tiempo de ciclo describe qué tan a menudo puede hacer referencia a ella. Suenan como la misma cosa, pero no lo son. Por ejemplo, si usted pide datos de un chip de DRAM con un tiempo de acceso de 50 ns, puede necesitar 100 ns antes de que pueda solicitar datos del mismo chip. Ello se debe a que los chips deben recobrarse internamente del acceso previo. Sin embargo, algunas tecnologías han mejorado el rendimiento cuando se está recuperando datos secuencialmente de una DRAM. En tales chips, los datos que inmediatamente después de aquellos accesados previamente, pueden recuperarse tan rápidamente como 10 ns.

Los tiempos de acceso y ciclo de memorias DRAM comerciales son más cortos que hace tan sólo algunos años, lo cuál significa que se pueden construir sistemas de memoria más rápidos. Pero también ha incrementado la velocidad de reloj de la CPU. El mercado de las computadoras caseras es un buen ejemplo. A inicios de la década de 1980, el tiempo de acceso de una DRAM comercial (200 ns) era menor que el ciclo de reloj de la IBM PC XT (4.77 MHz = 210 ns). Ello significa que la DRAM podía conectarse directamente a la CPU, sin preocuparse por rebasar al sistema de memoria. Pero a mitad de la década de 1980 se introdujeron modelos XT y AT más rápidos, con CPUs cuyos relojes superaban a los tiempos de acceso de la memoria comercial disponible. Había memorias más rápidas para quien estuviera dispuesto a pagar por ellas, pero los vendedores apostaron por vender computadoras que agregaban *estados de espera* al ciclo de acceso a la memoria. Los estados de espera son retrasos artificiales que hacen más lentos las referencias, de forma que la memoria parece empatarse con una

CPU más rápida -a cambio de una penalización. Sin embargo, esta técnica de agregar estados de espera comenzó a impactar significativamente el rendimiento alrededor de los 25 a 33 MHz. Hoy en día, las velocidades de las CPU están mucho más arriba que las de la DRAM.

La duración de un ciclo de reloj para las computadoras caseras comerciales ha cambiado de los 210 ns de una XT, a alrededor de 3 ns para una Pentium II a 300 MHz. Pero el tiempo de acceso para una DRAM comercial ha decrecido desproporcionadamente menos -de 200 ns a alrededor de 50 ns. El rendimiento del procesador se duplica cada 18 meses, mientras que el rendimiento de la memoria se duplica aproximadamente cada siete años.

La brecha de velocidad entre CPU y memoria es todavía mayor en el caso de las estaciones de trabajo. Algunos modelos tienen periodos de reloj tan cortos como 1.6 ns. ¿Cómo concilian los vendedores esta diferencia de velocidad entre CPU y memoria? La memoria en la supercomputadora Cray-1 empleaba SRAM que era capaz de mantenerse a la par de su ciclo de reloj de 12.5 ns. Usar SRAM en su memoria principal era una de las razones por las que la mayoría de las computadoras Cray requerían refrigeración líquida.

Desafortunadamente, no es práctico para un sistema de precio moderado confiar exclusivamente en la SRAM como almacenamiento. Como tampoco lo es fabricar sistemas económicos con almacenamiento suficiente usando sólo SRAM.

La solución es una jerarquía de memorias, formada por los registros del procesador, de uno a tres niveles de cache SRAM, una memoria principal DRAM y memoria virtual almacenada en medios tales como los discos. En cada punto de esta jerarquía de memoria se emplean trucos para lograr un uso óptimo de la tecnología disponible. En lo que resta de este capítulo examinaremos la jerarquía de memoria y su impacto sobre el rendimiento.

De cierta forma, con los procesadores actuales de alto rendimiento realizando cálculos tan rápidamente, la tarea del programador de alto rendimiento se convierte en administrar cuidadosamente la jerarquía de memoria. En cierto sentido, resulta un ejercicio intelectual útil pesar que los cálculos simples -tales como la suma y la multiplicación- son "infinitamente

rápidos", con el objeto de dar al programador una perspectiva correcta acerca del impacto de las operaciones de memoria sobre el rendimiento global del programa.

Registros

Cuando menos el estrato superior de la jerarquía de memoria, los registros de la CPU, operan tan rápido como el resto del procesador. El objetivo es mantener los operandos en los registros tanto como sea posible. Ello resulta especialmente importante para los valores intermedios usados durante cálculos largos, tales como:

```
X = G * 2.41 + A / W - W * M
```

Mientras calculamos el cociente de A entre W, debemos mantener almacenado el resultado de multiplicar G por 2.41. Sería una lástima tener que almacenar este resultado intermedio en memoria, para luego recargarlo unas pocas instrucciones más tarde. En cualquier procesador moderno con una optimización moderada, el resultado inmediato se almacena en un registro. Y como además el valor W se usa en dos cálculos, puede cargarse una vez y usarse dos, para eliminar el "desperdicio" que significaría otra operación de carga.

A partir de la década de 1970, los compiladores se han vuelto muy buenos en detectar este tipo de optimizaciones, y hacer un uso eficiente de los registros disponibles. Agregar más registros al procesador acarrea algún beneficio en el rendimiento, pero no es práctico agregar tantos como para almacenar los datos del problema completo. Así que debemos recurrir a una tecnología de memoria más lenta.

Caches

Si, partiendo de los registros, descendemos en la jerarquía de memoria, encontramos las caches. Se trata de pequeñas cantidades de SRAM que almacenan un subconjunto de lo contenidos de la memoria. La esperanza es que la cache tenga el subconjunto adecuado de memoria principal en el momento adecuado.

La arquitectura de la caché tuvo que cambiar conforme la duración del ciclo de los procesadores ha mejorado. Los procesadores son tan rápidos que ni siquiera los chips de SRAM son lo suficientemente rápidos. Ello ha conducido a un enfoque de cache multinivel con uno, o incluso dos, niveles de la misma implementadas como parte del procesador.[link] muestra la velocidad aproximada para acceder a la jerarquía de memoria de una DEC Alpha 21164 a 500 MHz.

Registros	2 ns
Nivel 1 en el chip	4 ns
Nivel 2 en el chip	5 ns
Nivel 3 fuera del chip	30 ns
Memoria	220 ns

Velocidades de acceso a la memoria en una DEC Alpha 21164.

Cuando puede encontrarse en una cache cada uno de los datos referenciados, se dice que se tiene una tasa de acierto del 100%. Generalmente, se considera que una tasa de acierto del 90% o superior es buena para una cache de Nivel 1 (L1). En la cache de Nivel 2 (L2), se

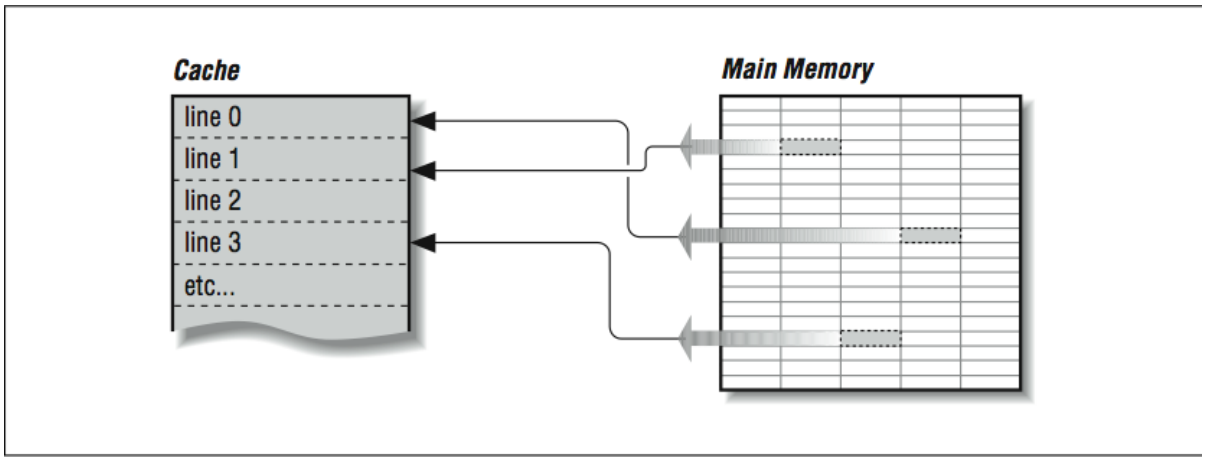
considera aceptable una tasa de acierto superior al 50%. De ahí hacia abajo, el rendimiento de la aplicación puede caer de forma vertiginosa.

Se puede caracterizar el rendimiento promedio de lectura de la jerarquía de memoria al examinar la probabilidad de que una carga particular se satisfaga en un nivel particular de la jerarquía. Por ejemplo, asumamos una arquitectura de memoria con una velocidad de cache L1 de 10 ns, una velocidad en L2 de 30 ns y una velocidad de memoria de 300 ns. Si una referencia a memoria dada se satisface mediante la cache L1 el 75% de las veces, 20% de ellas en la L2, y 5% del tiempo en la memoria principal, el rendimiento promedio de la memoria será:

```
(0.75 * 10 ) + ( 0.20 * 30 ) + ( 0.05 * 300 ) =
28.5 ns
```

Puede usted notar fácilmente por qué es tan importante tener una tasa de éxito de 90% o más en la cache L1.

Dado que una memoria cache almacena sólo un subconjunto de la memoria principal en un momento dado, es importante mantener un índice de cuáles áreas de la memoria principal están almacenadas actualmente en la cache. Para reducir la cantidad de espacio que debe dedicarse a seguir la pista de las áreas de memoria en cache, ésta se divide en un número de ranuras de igual tamaño, conocidas como *líneas*. Cada línea contiene cierto número de localidades secuenciales de memoria, generalmente de cuatro a dieciseis números enteros o reales. Mientras que los datos adentro de una línea vienen todos de la misma porción de memoria, otras líneas pueden contener datos de partes alejadas de su programa, o tal vez datos provenientes de los programas de alguien más, como en [link]. Cuando usted solicita algo de la memoria, la computadora comprueba si tales datos están disponibles en alguna de esas líneas de cache. Si es el caso, los datos se regresan con un retraso mínimo. Si no, puede que su programa se retrase un poco, mientras se carga una nueva línea de la memoria principal. Por supuesto, si se trajo nuevo contenido para una línea, el contenido de ésta debió primero desalojarse. Si tiene suerte, no será aquella que contenga los datos que necesitará justamente después.
Las líneas de cache pueden venir de diferentes partes de la memoria.



En los multiprocesadores (computadoras con varias CPUs), los datos escritos deben regresarse a la memoria principal, de forma que el resto de los procesadores puedan verlos, o se debe tener a todos los demás procesadores al tanto de la actividad de la cache local. Tal vez se requiera decirles que invaliden las líneas antiguas que contienen valores previos de la variable escrita, para evitar que accidentalmente usen datos viejos. a esto se le denomina mantener la *coherencia* entre caches diferentes. El problema se puede volver muy complejo en un sistema multiprocesador.[footnote] [link] describe la coherencia entre caches con mayor detalle.

Las caches son efectivas porque los programas a menudo exhiben características que ayudan a mantener alta la tasa de aciertos. Estas características se llaman localidad de referencia*espacial* y *temporal*; los programas frecuentemente hacen uso de datos e instrucciones que están cerca de otros datos e instrucciones, tanto en el espacio como en el tiempo. Cuando se carga una línea de cache de la memoria principal, no sólo contiene la información que causó el fallo de la cache, sino también algo de su información circundante. Hay buenas posibilidades de que la próxima vez que su programa necesite datos, éstos se encuentren en la línea de cache recién cargada o en alguna otra reciente.

Las caches trabajan mejor cuando un programa lee secuencialmente a través de la memoria. Asumamos que un programa está leyendo enteros de 32 bits con un tamaño de línea de cache de 256 bits. Cuando el programa hace referencia a la primera palabra en la línea de cache, debe esperar mientras dicha línea se carga desde la memoria principal. Las siete referencias a la

memoria subsecuentes se satisfacerán rápidamente desde la cache. Esto se llama *paso unitario* porque la dirección de cada elemento de datos sucesivo se incrementa en uno, y se usan todos la datos cargados en la cache. El siguiente ciclo funciona con un paso unitario:

```
DO I=1,1000000
  SUM = SUM + A(I)
END DO
```

Cuando un programa accede a una estructura de datos grande usando "pasos no unitarios", el rendimiento sufre porque se cargan datos en cache que no se usan. Por ejemplo:

```
DO I=1,1000000, 8
  SUM = SUM + A(I)
END DO
```

Este código carga la misma cantidad de datos y experimenta el mismo número de fallas en cache que el ciclo previo. Sin embargo, el programa necesita sólo una de las ocho palabras de 32 bits cargadas en la cache. Incluso aunque este programa realiza 1/8 de las sumas que el ciclo anterior, el tiempo que dilata en ejecutarse es aproximadamente el mismo que el otro, porque las operaciones de memoria dominan el rendimiento.

Aunque este ejemplo puede parecer un poco artificial, hay muchas situaciones en las cuales ocurren frecuentemente pasos no unitarios. Primero, cuando se carga en FORTRAN un arreglo bidimensional en memoria, los elementos sucesivos en la primera columna se almacenan secuencialmente, seguidos por los elementos de la segunda columna. Si el arreglo se procesa colocando la iteración de los renglones en el ciclo más interno, produce un patrón de referencias de pasos unitarios como el siguiente:

```
REAL*4 A(200,200)
DO J = 1,200
  DO I = 1,200
    SUM = SUM + A(I,J)
  END DO
END DO
```

Resulta interesante señalar que muy probablemente un programador en FORTRAN escribirá el ciclo (en orden alfabético) como sigue, produciendo un incremento no unitario de 800 bytes entre operaciones de carga sucesiva:

```
REAL*4 A(200,200)
DO I = 1,200
  DO J = 1,200
    SUM = SUM + A(I,J)
  END DO
END DO
```

Por esta razón, algunos compiladores pueden detectar este orden de ciclos subóptimo e invertirán el orden de los ciclos para lograr un mejor uso del sistema de memoria. Sin embargo, como veremos en [link], esta transformación de código puede producir resultados diferentes, y así usted deberá darle "permiso" al compilador para intercambiar esos ciclos en este ejemplo particular (o, tras haber leído este libro, simplemente haberlo codificado apropiadamente desde el comienzo).

```
while ( ptr != NULL ) ptr = ptr->next;
```

El siguiente elemento que se recuerda se basa en el contenido del elemento actual. Este tipo de ciclo salta por toda la memoria sin un patrón particular.

Se le conoce como *caza de apuntadores*, y no existe una forma acertada de mejorar el rendimiento de este código.

Un tercer patrón que se encuentra a menudo en cierto tipo de códigos se conoce como acopio (o dispersión), y ocurre en ciclos como:

```
SUM = SUM + ARR ( IND(I) )
```

donde el arreglo IND contiene desplazamientos dentro del arreglo ARR. De nuevo, tal como sucedió en la lista ligada, el patrón exacto de referencias a memoria sólo se conoce a tiempo de ejecución, cuando también se conocen los valores almacenados en el arreglo IND. Algunos sistemas de propósito especial tienen soporte de hardware especializado para acelerar esta operación en particular.

Organización de la Cache

El proceso de aparear las localidades de memoria con las líneas de cache se llama *mapeo*. Por supuesto, dado que la cache es menor que la memoria principal, tendrá usted que compartir las mismas líneas de cache entre distintas localidades de memoria. En las caches, cada línea mantiene un registro de las direcciones de memoria (conocido como la *etiqueta*) a las que representa, y tal vez de cuándo se usaron por última vez. La etiqueta se usa para seguir la pista a cuál área de memoria está almacenada en una línea particular de cache.

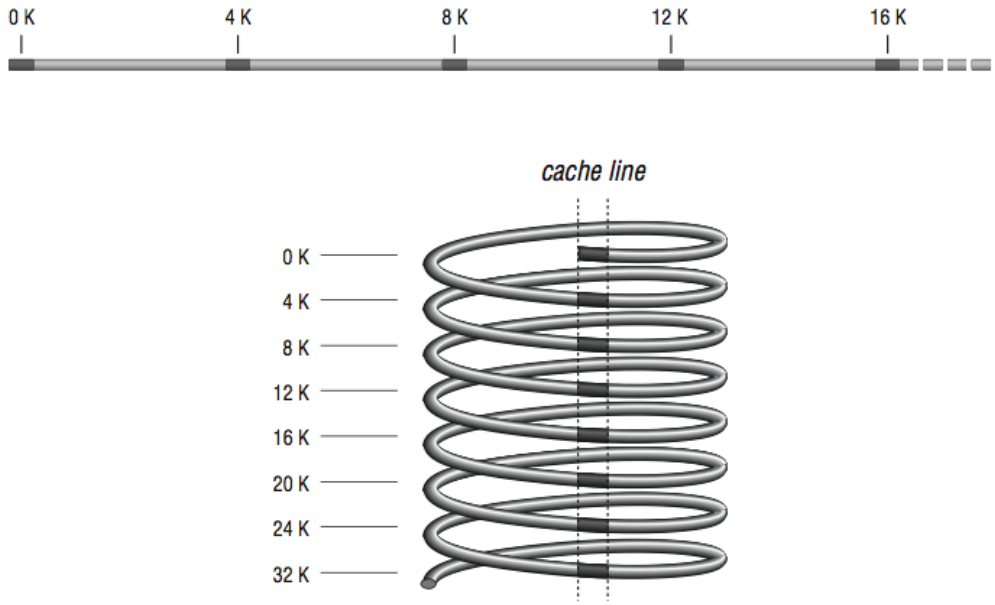
La forma en que las localidades de memoria (etiquetas) se mapean a líneas de cache puede tener un efecto benéfico en la forma en que se ejecuta su programa, porque si dos localidades de memoria utilizadas intensamente se mapean a la misma línea de cache, la tasa de fallos será mayor de lo que usted quisiera. Las caches pueden organizarse de varias maneras: mapeadas directamente, completamente asociativas y asociativas en conjunto.

Caches Mapeadas Directamente

El mapeo directo, tal como se muestra en [link], es el algoritmo más sencillo para decidir cómo mapear la memoria en la cache. Digamos, por ejemplo, que su computadora tiene 4 KB de cache. En un esquema de mapeo directo, la localidad de memoria 0 se mapea en la localidad 0 de cache, así como las localidades 4K, 8K, 12K, etc. En otras palabras,la memoria se mapea en bloques del mismo tamaño que la cache. Otra forma de verlo es imaginar un resorte de metal con una línea de gis marcada a toda su longitud. Cada vuelta alrededor del resorte, se encuentra la línea de gis en el mismo lugar, módulo la longitud de circunferencia del resorte. Si éste es muy largo, la línea de gis cruza muchas vueltas de la bobina, análogo a como sucede con una memoria grande con muchas localidades mapeándose a la misma línea de cache.

Los problemas devienen cuando a tiempo de ejecución se alternan referencias a memoria en un punto de la cache mapeada directamente, a la misma línea de cache. Cada referencia causa un fallo de caché y reemplaza la entrada que recién se había reemplazado, causando mucha sobrecarga. El

término coloquial para este fenómeno es *thrashing*. Cuando ocurre con demasiada frecuencia, una cache puede ser más una carga que una ayuda, porque cada falla de la caché requiere que una línea completa se llene nuevamente -una operación que mueve más datos que lo que supondría meramente satisfacer directamente la referencia desde la memoria principal. Es fácil construir un caso patológico que cause thrashing en una cache de 4 KB mapeada directamente:
Muchas direcciones de memoria mapeadas a la misma línea de cache.




```
REAL*4 A(1024), B(1024)
COMMON /STUFF/ A,B
DO I=1,1024
  A(I) = A(I) * B(I)
END DO
END
```

Ambos arreglos, A y B, tiene exactamente 4 KB de almacenamiento, y su inclusión juntos en `COMMON` garantiza que los arreglos inician exactamente con 4 KB de distancia en memoria. En una cache de 4KB con mapeo

directo, la misma línea que se usa para A(1) se emplea también para B(1), y lo mismo sucede con A(2) y B(2), etc., de tal suerte que las referencias alternadas causan repetidas fallas de cache. Para solucionarlo, debe bien sea ajustar el tamaño del arreglo A, o poner algunas otras variables adentro de `COMMON`, en medio de ambas. Por esta razón uno debe generalmente evitar tamaños de arreglo cercanos a potencias de dos.

Cache Completamente Asociativa

En el extremo opuesto de una cache directamente mapeada se encuentra una *cache completamente asociativa*, donde cualquier localidad de memoria puede mapearse en cualquier línea de cache, independientemente de la dirección de memoria. Las caches completamente asociativas obtienen su nombre del tipo de memoria usada para construirlas -memoria asociativa- que es como la memoria regular, excepto por el hecho de que cada celda sabe algo acerca de los datos que contiene.

Cuando el procesador busca un dato, pregunta simultáeamente a todas las líneas de cache si alguna de ellas lo tiene. La línea que contiene dicho dato alza su mano y dice "Lo tengo"; si ninguna contesta, ocurre una falla de caché. Entonces viene la cuestión de cuál línea de cache debe reemplazarse con los nuevos datos. En vez de mapear las localidades de memoria a las líneas de cache mediante un algoritmo, como sucede en el cache directamente mapeado, el sistema de memoria puede pedir a las líneas de cache completamente asociativo que elijan entre ellas cuáles localidades de memoria representarán. Usualmente la línea que menos se ha usado recientemente es la que se sobreescribe con los nuevos datos. La presunción es que si los datos no han sido empleados durante un tiempo, es menos posible que se requieran en el futuro.

Las caches completamente asociativas presentan una utilización superior, cuando se comparan con caches directamente mapeadas. Es difícil encontrar ejemplos de programas del mundo real que causen thrashing en un cache completamente asociativo. El precio de las caches completamente asociativas es muy alto, en términos de tamaño, precio y velocidad. Las caches asociativas que realmente existen tienden a ser pequeñas.

Cache Asociativa en Conjunto

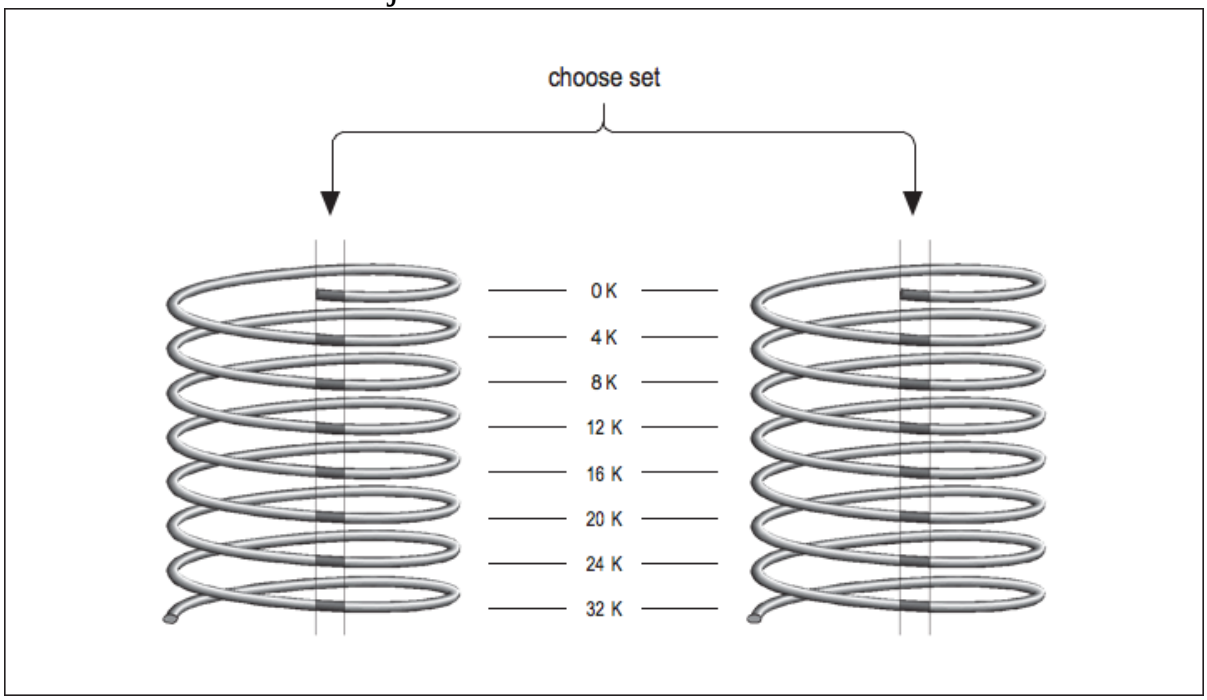
Ahora imagine que tiene dos caches directamente mapeadas, sentadas una junto a otra en una sola unidad de cache, como se muestra en [link]. Cada localidad de memoria corresponde a una línea particular de cache en cada una de las dos caches directamente mapeadas. Aquella que elija reemplazar durante una falla de cache queda sujeta a decisión dependiendo de cuál línea fue usada hace más tiempo -en la misma forma en que se decide en una cache completamente asociativa, excepto que ahora sólo hay dos opciones. A esto se le llama una *cache asociativa en conjunto*, y generalmente vienen en dos y cuatro bancos de cache separados. Se les llama caches asociativas de conjuntos de *dos vías* y *cuatro vías*, respectivamente. Por supuesto, cada tipo tiene sus ventajas e inconvenientes. Una cache asociativa en conjunto es más inmune al thrashing que una cache directamente mapeada del mismo tamaño, porque para cada mapeo de una dirección de memoria en una línea de cache, hay dos o más opciones de destino. Sin embargo, la belleza de una cache directamente mapeada es que resulta fácil de implementar y, si se hace lo suficientemente grande, rinde prácticamente igual de bien que un diseño asociativo en conjunto. Su máquina puede contener múltiples caches para varios propósitos distintos. He aqui un pequeño programa que causa thrashing en una cache asociativa en conjunto de dos vías y 4 KB:

```
REAL*4 A(1024), B(1024), C(1024)
COMMON /STUFF/ A,B,C
DO I=1,1024
  A(I) = A(I) * B(I) + C(I)
END DO
END
```

Al igual que el programa anterior, éste obliga a repetir los accesos a las mismas líneas de caché, excepto que ahora hay tres variables contendientes a ser elegidas para la asignación del mismo conjunto, en lugar de dos. De nuevo, la forma de solucionarlo consiste en cambiar el tamaño de los

arreglos, o insertar algo entre ellos en `COMMON`. Por cierto, si accidentalmente construye usted un programa como este, le resultará difícil detectarlo -más allá de sentir que el programa se ejecuta algo lento. Pocos proveedores proporcionan herramientas para medir las fallas de cache.

Cache asociativa en conjunto de dos vías.



Cache de Instrucciones

Hasta el momento hemos pasado por alto los dos tipos de información que se espera encontrar en una cache ubicada entre la memoria y la CPU: instrucciones y datos. Pero si piensa en ello, la demanda de datos se encuentra separada de la de instrucciones. En los procesadores superescalares, por ejemplo, es posible ejecutar una instrucción que causa una falla en la cache de datos junto con otras instrucciones que no requieren datos de la cache en absoluto, es decir, que operan sobre los registros. No parece justo que una falla de cache en una referencia a datos en una instrucción deba evitarle recuperar otras instrucciones por que la cache está atada. Además, una cache depende localmente de la referencia entre bits de datos y otros bits de datos o instrucciones y otras instrucciones, pero ¿qué clase de interdependencia existe entre instrucciones y datos? Parece imposible que las instrucciones extraigan datos perfectamente útiles de la

cache, o viceversa, con completa independencia de la localidad de la referencia.

Muchos diseños desde la década de 1980 usan una sola cache tanto para instrucciones como para datos. Pero los diseños más nuevos están empleando lo que se conoce como la *Arquitectura de Memoria Harvard*, donde la demanda de datos se separa de la demanda de instrucciones.

La memoria todavía sigue siendo un único repositorio grande, pero estos procesadores tienen caches separadas de datos e instrucciones, posiblemente con diseños diferentes. Al proporcionar dos fuentes independientes para datos e instrucciones, la tasa de información agregada proveniente de la memoria se incrementa, y la interferencia entre los dos tipos de referencias a memoria se minimiza. Además, las instrucciones generalmente tienen un nivel de localidad de referencia extremadamente alto, debido a la naturaleza secuencial de la mayoría de los programas. Como las caches de instrucciones no tienen que ser particularmente grandes para ser efectivas, una arquitectura típica consiste en tener dos caches L1 separadas para datos e instrucciones, y tener una cache L2 combinada. Por ejemplo, el PowerPC 604e de IBM/Motorola tiene caches L1 separadas de 32K de cuatro vías para instrucciones y datos, y una cache L2 combinada.

Memoria Virtual

La memoria virtual desacopla las direcciones usadas por el programa (direcciones virtuales) de las direcciones donde realmente está almacenado el dato en memoria (direcciones físicas). Su programa ve sus direcciones comenzando desde 0 y avanzando hasta algún número grande, pero las direcciones físicas a las que realmente están asignadas pueden ser muy diferentes. Esto proporciona cierto grado de flexibilidad, pues permite a todos los procesos creer que tienen el sistema de memoria completo para ellos. Otro rasgo de los sistemas de memoria virtual es que dividen la memoria de sus programas en *páginas* — fragmentos. Los tamaños de página varían entre 512 bytes y 1 MB o más grande, dependiendo de la máquina. Las páginas no tienen por qué estar físicamente contiguas, aunque su programa las vea de esta forma. Al estar separados en páginas, los programas son más fáciles de acomodar en memoria, o mover porciones de los mismos hacia el disco.

Tablas de Páginas

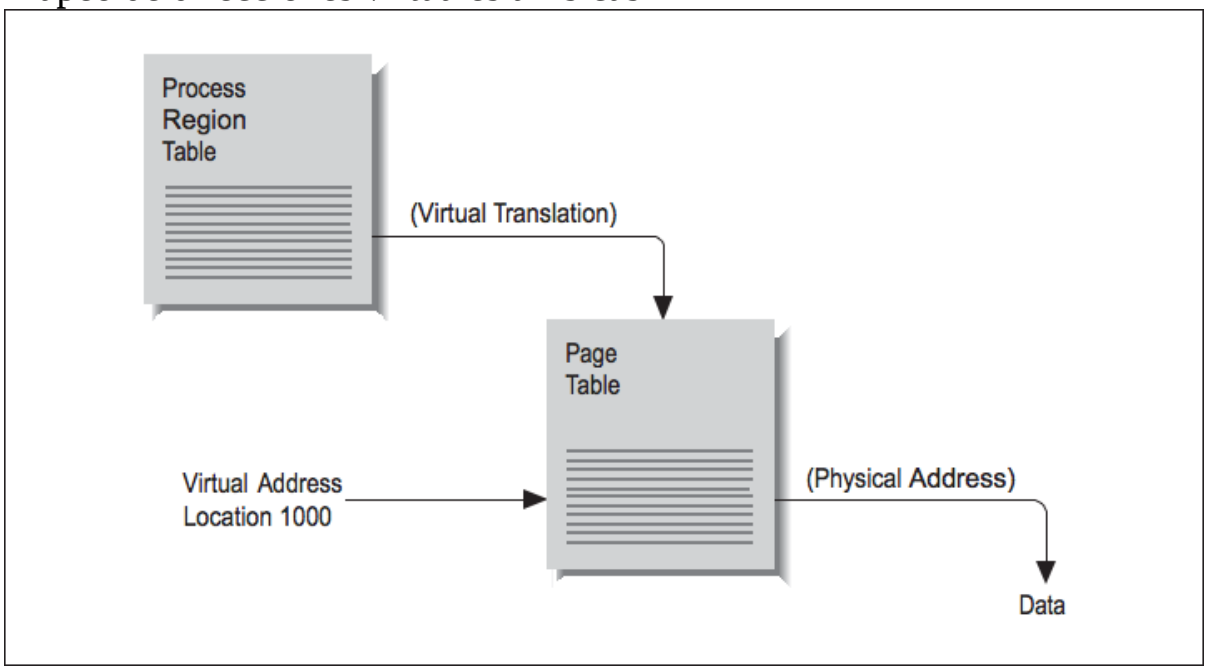
Digamos que su programa solicita una variable almacenada en la localidad 1000. En una máquina con memoria virtual, no hay una correspondencia directa entre la idea que su programa tiene acerca de dónde se encuentra la localidad 1000, y la que tiene el sistema de memoria. Para hallar dónde está almacenada realmente la variable, la dirección tiene que traducirse de virtual a física. El mapa que contiene tales traducciones se llama una *tabla de páginas*. Cada proceso tiene asociadas varias tablas de páginas, correspondientes a distintas regiones, tales como los segmentos de texto y datos del programa.

Para entender cómo funciona la traducción de direciones, imagine el siguiente escenario: en algún momento su programa solicita los datos almacenados en la localidad 1000. [link] muestra los pasos necesarios para completar la recuperación de esos datos. Al elegir la localidad 1000, usted ha identificado en qué región de memoria recae esa referencia, y ésta identifica cuál tabla de página debe usarse. Por tanto, la localidad 1000 ayuda al procesador a elegir una entrada en la tabla. Por ejemplo, si el

tamaño de página es de 512 bytes, 1000 cae en la segunda página (las páginas ocupan los rangos 0-511, 512-1023, 1024-1535, etc.)

Por tanto, la segunda entrada en la tabla debe almacenar la dirección de la página que contiene el valor en la localidad 1000.

Mapeo de direcciones virtuales a físicas



El sistema operativo almacena virtualmente las direcciones de la tabla de páginas, por lo que tendrá que realizar una traducción virtual a física para localizar la tabla en la memoria. Una traducción de virtual a física más, y finalmente tenemos la verdadera dirección de la localidad 1000. La referencia a memoria puede completarse, y el procesador puede continuar ejecutando su programa.

Buffer de Traducción de Direcciones

Como puede ver, la traducción de una dirección a través de una tabla de páginas resulta bastante complicada. Requiere dos búsquedas en tablas (tal vez tres) para encontrar nuestros datos. Si cada referencia a memoria fuera así de complicada, las computadoras con memoria virtual tendrían rendimientos pésimos. Afortunadamente, la localidad de referencias causa que las traducciones de direcciones virtuales se agrupen; un programa puede repetir el mismo mapeo de páginas virtuales millones de veces por

segundo. Y allí donde tenemos un uso repetido de los mismos datos, podemos aplicar una cache.

Todas las máquinas modernas con memoria virtual tienen una cache especial llamada *buffer de traducción de direcciones* (TLB por sus siglas en inglés) para la conversión de direcciones de memoria virtuales a físicas. Las dos entradas a la TLB son enteros que identifican tanto al programa que hace la solicitud a memoria, como la página virtual solicitada. Desde la salida surge un apuntador al número de página física. Ingresan direcciones virtuales, salen direcciones físicas. Las búsquedas en la tabla ocurren en paralelo con la ejecución de la instrucción, así que si la dirección de los datos está en la TLB, las referencias a la memoria ocurrirán de forma muy rápida.

Como otros tipos de caches, la TLB tiene un tamaño limitado. No contiene entradas suficientes para manejar todas las posibles traducciones de direcciones virtuales a físicas para todos los programas que pueda usted ejecutar en su computadora. Los depósitos más grandes de traducciones de direcciones se mantienen fuera en memoria, en las tablas de páginas. Si su programa solicita una traducción de dirección virtual a física, y no existe una entrada en la TLB, sufrirá usted una *falla TLB*. Puede que la información requerida deba generarse (es decir, crearse una nueva página), o puede que deba recuperarse de la tabla de páginas.

La TLB es buena por la misma razón que lo son otros tipos de cahes: reducen el costo de hacer referencias a memoria. Pero como las otras caches, existen casos patológicos en los cuales la TLB puede fallar en la entrega de un valor. El caso más sencillo de construir es uno donde cada referencia a memoria que haga su programa cause una falla TLB:

```
REAL X(10000000)
COMMON X
DO I=0,9999
  DO J=1,10000000,10000
    SUM = SUM + X(J+I)
  END DO
```

```
END DO
```

Asumamos que el tamaño de página de la TLB de su computadora es menor a 40 KB. Cada vez que se recorre el ciclo interno en el código de ejemplo anterior, el programa solicita datos que están alejadas 4 bytes*10,000 = 40,000 bytes respecto a la última referencia. Esto es, cada referencia cae en una página de memoria diferente. Ello causa 1000 fallas TBL en el ciclo interior, repetido 1001 veces, para un total de cuando menos un millón de fallas TLB. Para hacer todavía más grave el problema, está garantizado que cada referencia cause también una falla del cache de datos. Es claro que nada debe comenzar con un ciclo como el de arriba. Pero suponiendo que el ciclo era del todo bueno para usted, la versión reestructurada del código siguiente atraviesa la memoria como un cuchillo caliente la mantequilla:

```
REAL X(10000000)
COMMON X
DO I=1,10000000
  SUM = SUM + X(I)
END DO
```

El ciclo revisado tiene saltos unitarios, y las fallas TLB ocurren sólo de modo muy ocasional. Usualmente no es necesario afinar explícitamente los programas para que hagan un buen uso de la TLB. Una vez que el programa se modifica para ser "amigable con la cache", muy probablemente se haya afinado también para ser amigable con la TLB.

Dado que hay beneficios de rendimiento derivados de mantener muy pequeña la TLB, a menudo cada una de sus entradas contiene un campo de longitud. Una sola entrada en la TLB puede tener cerca de un megabyte de longitud, y puede usarse para traducir direcciones almacenados en múltiples páginas de memoria virtual.

Fallos de Página

Cada elemento en la tabla de páginas también contiene otra información acerca de la página que representa, incluyendo banderas que indican si la traducción es válida, si la página asociada puede modificarse, y alguna información que describe cómo deben iniciarse las páginas nuevas. Aquellas referencias a páginas que no están marcadas como válidas se denominan *fallos de página*.

Partiendo del peor escenario posible, digamos que su programa solicita una variable en una localidad de memoria particular. El procesador lo busca en la cache y encuentra que no está ahí (falla de cache), lo cual significa que debe cargarse desde la memoria. Lo siguiente que hace es ir a la TLB para encontrar la localidad física del dato en memoria, y descubre que o hay una entrada en la TLB (una falla TLB). Entonces trata de consultar la tabla de páginas (y rellenar nuevamente la TLB), pero encuentra que o bien no hay una entrada para esta página en particular, o que dicha página se envió a disco (ambos casos son fallos de página). Cada paso en la jerarquía de memoria ha complicado su solicitud. Debe crearse una nueva página y posiblemente, dependiendo de las circunstancias, recargarla a partir del disco.

Pero independientemente de que tomen mucho tiempo, los fallos de página no son errores. Incluso bajo condiciones óptimas, cada programa sufre cierto número de fallos de página. Escribir una variable por primera vez o llamar a una subrutina que no se había invocado previamente, causarán un fallo de página. Si no lo había pensado anteriormente puede parecerle sorpresivo. Existe la ilusión de que su programa completo está presente en memoria desde el inicio, pero puede que ciertas porciones jamás se carguen. No hay razón para hacerle espacio a una página a cuyos datos nunca se hace referencia, o cuyas instrucciones no se ejecutan. Sólo se crean o se traen del disco aquellas páginas que se necesitan para ejecutar el trabajo. [footnote] El término adecuado para referirse a esto es demanda de páginas.

El repositorio de páginas de memoria física está limitado, porque la memoria física también lo está, de modo que en una máquina en la que muchos programas cabildean por el espacio, ocurrirán una gran cantidad de fallos de página. Ello se debe a que las páginas de memoria física están reciclándose continuamente para otros propósitos. Sin embargo, cuando

tiene la máquina sólo para usted, y hay menos demanda de memoria, las páginas asignadas tienden a mantenerse por algún tiempo. En resumen, puede usted esperar menos fallos de página en una máquina tranquila. Un truco que debe recordar, si siempre termina usted trabajando para un proveedor de computadoras: ejecute siempre benchmarks cortos dos veces. En algunos sistemas, el número de fallos de página decaerá. Ello se debe a que la segunda ejecución encuentra páginas dejadas en memoria por la primera, y usted no tiene que pagar el precio de los fallos de página nuevamente.[footnote]
El dispositivo de disco identifica las páginas de texto y los números de bloque de los que procede.

El espacio de paginación (espacio de intercambio) en el disco es la última y más lenta de las piezas de la jerarquía de memoria en la mayoría de las máquinas. En el peor escenario vimos cómo puede enviarse una referencia a memoria hacia un medio más lento y con un rendimiento inferior, antes de que finalmente se satisfaga la solicitud. Si se regresa sobre sus pasos, podrá ver que el espacio de paginación en disco tiene la misma relación con la memoria principal, que ésta respecto a la cache. También se aplican las mismas clases de optimización, y la localidad de las referencias es importante. Puede ejecutar programas mayores que la memoria principal de su máquina, pero a veces a costa de una gran caída del rendimiento. Cuando revisemos las optimizaciones de memoria en [link], nos concentraremos en mantener en actividad las partes más rápidas del sistema de memoria, y evitar las más lentas.

Mejorando el Rendimiento de la Memoria

Dada la importancia, en el área del cómputo de alto rendimiento, del desempeño del subsistema de memoria de una computadora, se han usado muchas técnicas para tratar de mejorarlo. Sus dos atributos más importantes son el *ancho de banda* y la *latencia*. Ciertos diseños de sistemas de memoria mejoran uno a expensas del otro, mientras que otros impactan positivamente tanto en el ancho de banda como en la latencia. El ancho de banda generalmente se enfoca en la mejor tasa de transferencia del sistema de memoria en estado estacionario, lo que usualmente se mide durante la ejecución de un ciclo largo de avance unitario, que lee o lee y escribe la memoria. [footnote] La latencia es una medida del rendimiento de un sistema de memoria, en el peor caso conforme mueve una pequeña cantidad de datos (como por ejemplo una palabra de 32 o 64 bits) entre el procesador y la memoria. Ambos son importantes porque son parte sustancial de muchas aplicaciones de alto rendimiento.
Véase la sección FLUJO del documento=""/>Capítulo 15 para una revisión de las medidas del ancho de banda de la memoria.

Como los sistemas de memoria se dividen en componentes, hay valores de ancho de banda y latencia de diferentes órdenes de magnitud entre los distintos componentes, como se muestra en [link]. La tasa de ancho de banda entre una cache y la CPU será más alta que el ancho de banda entre la memoria principal y la cache, por ejemplo. Además, puede haber muchas caches y rutas a la memoria. Usualmente, el valor pico del ancho de banda citado por los vendedores es la velocidad entre la cache de datos y el procesador.

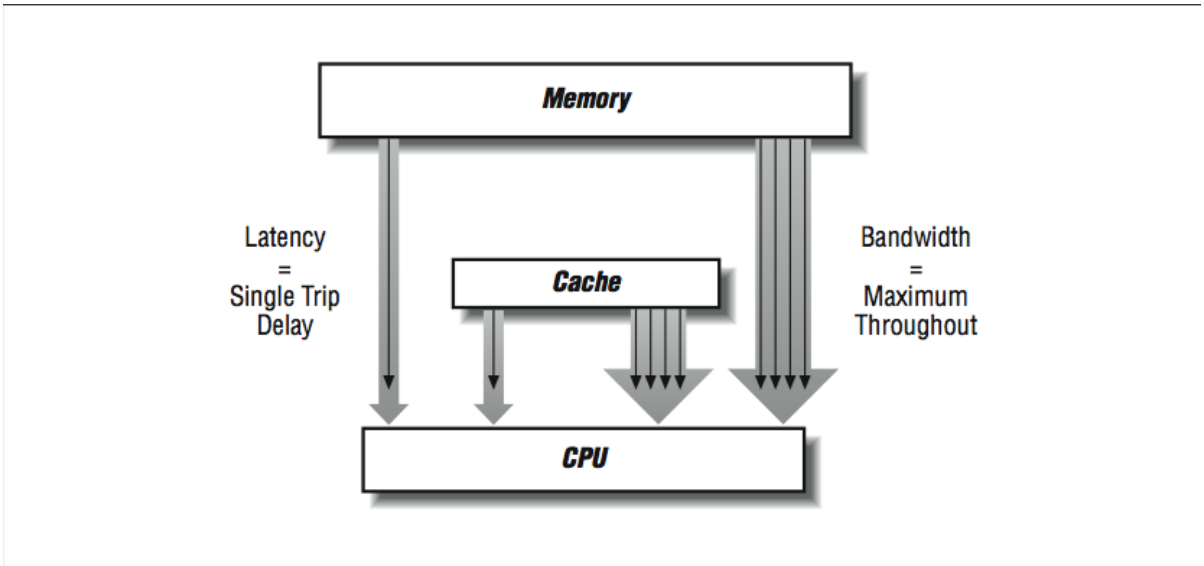
En el resto de esta sección, revisaremos las técnicas para mejorar la latencia, el ancho de banda o ambos.

Caches Grandes

Como mencionamos a inicios de este capítulo, la disparidad entre las velocidades de la CPU y la memoria está creciendo. Si lo observa de cerca, verá a los distribuidores innovando en varios aspectos. ¡Ofrecen algunas estaciones de trabajo con caches de datos de 4 MB! Es más que los sistemas

de memoria principal de las máquinas de apenas hace algunos años. Con una cache lo suficientemente grande, un conjunto de datos pequeño (o incluso moderadamente grande) puede caber en ella completamente, y lograr un rendimiento increíblemente bueno. Fíjese muy bien en este aspecto cuando pruebe hardware nuevo. Cuando su programa se torna demasiado grande para la cache, el rendimiento caerá considerablemente, tal vez en un factor de 10 o más, dependiendo de los patrones de acceso a la memoria. Resulta interesante señalar que un incremento en el tamaño de la cache por parte de los distribuidores puede volver obsoleto un benchmark.

Un sistema de memoria sencillo



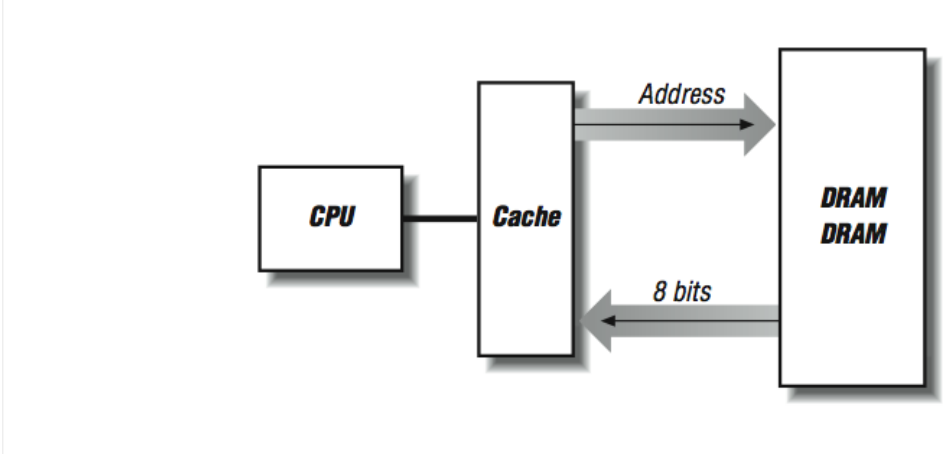
Hasta 1999, el benchmark Linpack 100x100 fue probablemente la prueba más respetada para determinar el rendimiento promedio comparativo de una amplia variedad de aplicaciones. En 1992, IBM introdujo la RS-6000, con una cache suficientemente grande para contener completa la matriz de 100x100 durante toda la duración de la prueba. Por vez primera, una estación de trabajo presentaba un rendimiento del mismo orden que las supercomputadoras. En un sentido, con la estructura de datos completa contenida en una cache SRAM, la RS-6000 operaba como una supercomputadora vectorial Cray. El problema era que la Cray podía mantener y memorar el rendimiento con matrices de 120x120, mientras que la RS-6000 sufría de una caída de rendimiento significativo con este aumento del tamaño de la matriz. Pronto, todos los otros vendedores de estaciones de trabajo introdujeron caches de tamaño similar, y la prueba

Linpack 100x100 dejó de ser útil como un indicador del rendimiento promedio de una aplicación.

Sistemas de Memoria más Anchos

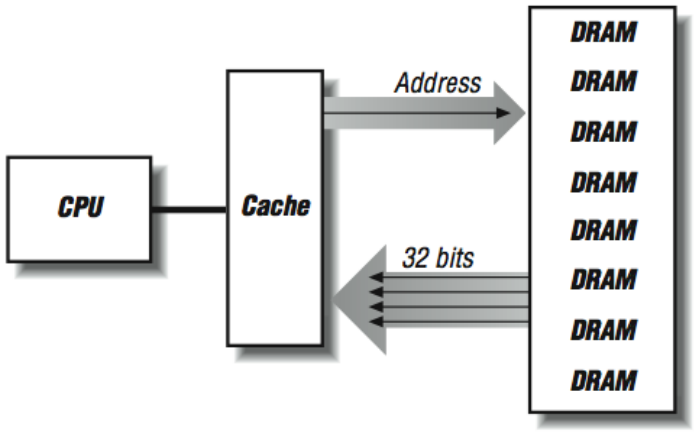
Considere lo que sucede cuando una línea de cache se rellena desde la memoria: se leen localidades de memoria consecutivas desde la memoria principal, para llenar con ellas localidades consecutivas en la línea de cache. El número de bytes transferidos dependen de cuan grande es la línea - cualquier cosa entre 16 bytes y 256 bytes o más. Queremos que esta operación de relleno se lleve a cabo rápidamente porque hay una instrucción atorada en la pipeline, o tal vez el procesador está esperando más instrucciones. En [link], si tenemos dos chips DRAM que nos proporcionan 4 bits de datos cada 100 ns (recuerde el tiempo de ciclo), llenar una línea de cache de 16 bytes toma 1600 ns.

Sistema de memoria angosto



Una forma de acelerar la operación de llenado de la línea de cache consiste en "ensanchar" el sistema de memoria, como se muestra en [link]. En vez de tener dos renglones de DRAMs, creamos múltiples renglones. Ahora en cada ciclo de 100 ns obtenemos 32 bits contiguos, y nuestra línea de cache se llena cuatro veces más rápido.

Sistema de memoria ancho



Podemos mejorar el rendimiento de un sistema de memoria, incrementando el ancho del mismo hasta que iguale la longitud de la línea de cache, momento en que podemos llenar la línea completa en un solo ciclo de memoria. En la serie de sistemas SGI Power Challenge el ancho de memoria es de 256 bits. El lado negativo de un sistema de memoria más ancho es que debe agregarse la DRAM en múltiplos enteros. En muchas estaciones de trabajo y computadoras personales modernas, la memoria se expande mediante módulos de memoria de una sola línea (SIMM por sus siglas en inglés), y dichos SIMMs son actualmente de 30, 72 o 168 patillas, cada uno de los cuales está hecho de varios chips DRAM listos para ser instalados en un subsistema de memoria.

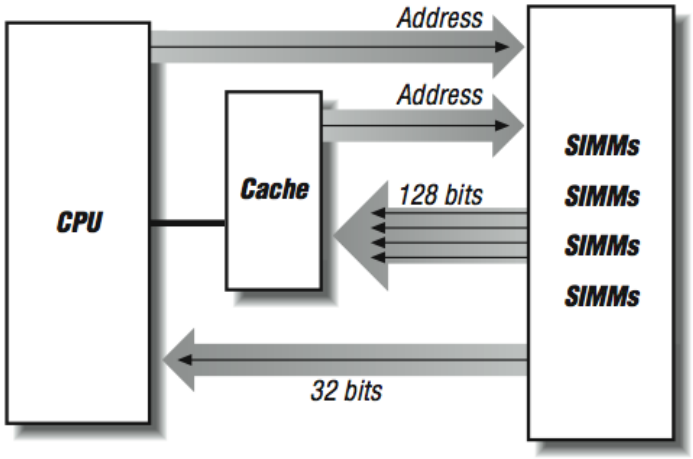
Evitando la Cache

Es interesante resaltar que casi hemos ocupado un capítulo completo a explicar lo buena que es la cache para las computadoras de alto rendimiento, y ahora vamos a evitar la cache para mejorar el rendimiento. Como se mencionó con anterioridad, algunos tipos de procesamiento resultan en incrementos no unitarios (o rebotes) a lo largo de la memoria. Estos tipos de patrones de referencia a memoria presentan el peor caso posible de comportamiento en arquitecturas basadas en cache. Este tipo de patrones de referencias es el que ve mejorado su rendimiento evitando la cache. La imposibilidad de soportar este tipo de computación continúa siendo un área donde las supercomputadoras tradicionales pueden comportarse peor que los procesadores RISC de alta velocidad. Por esta

razón, los procesadores RISC que abordan seriamente el procesamiento numérico suelen tener instrucciones especiales que evitan el uso de la memoria cache; los datos se transfieren directamente entre el procesador y el sistema de memoria principal. [footnote] En [link] tenemos cuatro bancos de SIMMs que pueden llenar la memoria cache a razón de 128 bits por cada ciclo de memoria de 100 ns. Recuerde que los datos están disponibles tras 50 ns, pero no podemos obtener más datos hasta que la DRAM se refresque, de 50 a 60 ns después. Sin embargo, si estamos realizando cargas de 32 bits en incrementos no unitarios y tenemos la capacidad de evitar la cache, cada carga quedará satisfecha desde alguno de los cuatro SIMMs en 50 ns. Mientras ese SIMM se refresca, puede ocurrir otra carga desde cualesquiera de los otros tres SIMMs en 50 ns. En una mezcla aleatoria de cargas no unitarias hay una probabilidad del 75% de que la siguiente carga caiga en una DRAM "fresca". Si la carga recae en un banco mientras se está refrescando, simplemente tiene que esperar hasta que se complete el refresco.
por cierto, muchas máquinas tienen espacios de memoria sin cache para sincronización de procesos y registros de E/S. Sin embargo, las referencias a memoria en tales localidades evitan la cache por causa de la dirección elegida, no necesariamente por la instrucción elegida.

Una ventaja extra de evitar la cache es que los datos no requieren moverse a través de la cache de SRAM. Esta operación puede agregar entre 10 y 50 ns al tiempo de carga para una palabra simple. Ello también evita tener que invalidar los contenidos de una línea de cache completa.

Agregar una forma de evitar la cache, incrementar el ancho de memoria del sistema y agregar bancos incrementa el costo de un sistema de memoria. Los fabricantes de sistemas de cómputo toman una decisión de índole económica acerca de cuántas de estas técnicas requieren aplicar para obtener el rendimiento suficiente en su sistema y procesador particulares. De este modo, conforme crece la velocidad del procesador, deben agregar más de estas características del sistema de memoria a sus equipos para mantener el balance entre las velocidades del procesador y el sistema de memoria.
Evitando la cache



Sistemas de Memoria Intercalados y Entubados

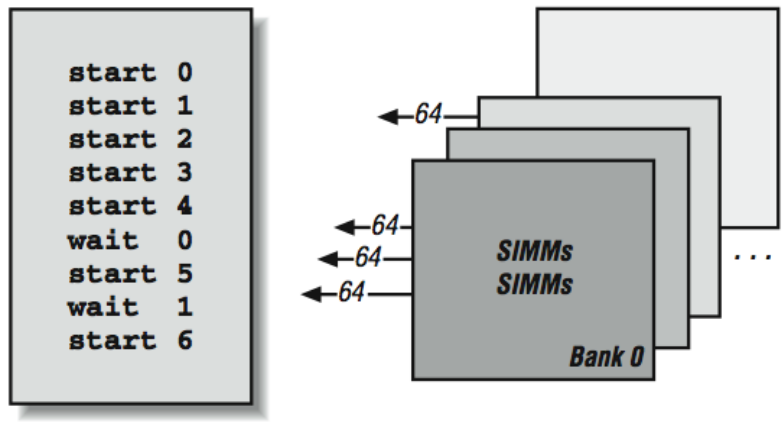
Las supercomputadoras vectoriales, tales como la CRAY Y/MP y la Convex C3, son máquinas que dependen de sistemas de memoria multibanco para lograr alto rendimiento. En particular la C3 tiene un sistema de memoria con intercalación de hasta 256 vías. Cada intercalado (o banco) tiene 64 bits de anchura. Se trata de un sistema de memoria caro de construir, pero tiene algunas características de rendimiento muy atractivas. Tener un gran número de bancos ayuda a reducir las posibilidades de accesos repetidos al mismo banco de memoria. Pero si se da el caso de acertar dos veces al mismo renglón, la penalización es un retraso de cerca de 300 ns -un tiempo largo para una máquina con un tiempo de reloj de 16 ns. Pero cuando las cosas salen bien, realmente van muy bien.

Sin embargo, no es suficiente tan sólo con tener un gran número de bancos para alimentar un prosador a 16 ns usando una DRAM a 50 ns. Además del intercalado, el sistema de memoria también requiere entubarse. Esto es, la CPU debe comenzar la segunda, tercera y cuarta cargas antes de haber recibido los resultados de la primera, como se muestra en [link]. Luego cada vez que recibe los resultados del banco "n", debe comenzar la carga del banco "n+4" para mantener alimentada la tubería. De esta forma, tras un breve retraso en el arranque, las cargas se completan cada 16 ns y así el sistema de memoria parece operar a la velocidad de reloj de la CPU. El

enfoque de memoria entubada se facilita por los registros vectoriales de 128 elementos en el procesador C3.

Usando hardware recolector/dispersor, también pueden entubarse operaciones de incremento no unitario. La única diferencia de las operaciones de incremento no unitario es que no se accede a los bancos en orden secuencial. Con un patrón aleatorio de referencias a memoria, es posible reacceder al banco de memoria antes de que el acceso previo se refresque completamente. Esto se llama *detención de banco.*.

Sistema de memoria multibanco



Diferentes patrones de acceso están sujetos a detenciones de banco de severidad variable. Por ejemplo, acceder a una palabra de cada cuatro en un sistema de memoria de ocho bancos puede estar sujeto a detenciones de banco, aunque la recuperación será rápida. Puede ser que hacer referencias a una palabra de cada dos no haga experimentar detenciones de banco en absoluto; cada banco puede haberse recuperado para el momento en que llegue la siguiente referencia; depende de las velocidades relativas del procesador y del sistema de memoria. De seguro que los patrones de acceso irregular encontrarán algunas detenciones de banco.

Además del peligro de la detención de banco, las referencias a una sola palabra que se realizan directamente a un sistema de memoria multibanco acarrean una latencia mayor que los accesos (exitosos) a memoria cache. Ello se debe a la referencias que se hacen a la memoria exterior, más lenta

que la cache, e igual puede haber pasos adicionales de traducción de direcciones. Sin embargo, las referencias a la memoria en bancos está entubada. Tan pronto como las referencias han iniciado por adelantado lo suficientemente bien, pueden ocurrir simultáneamente muchas referencias multibanco entubadas, que le proporcionan una buena tasa de rendimiento.

El sistema CDC-205 ejecutaba operaciones vectoriales en un estilo memoria a memoria, usando un conjunto explícito de entubamientos de memoria. Este sistema proporcionaba un rendimiento superior en cálculos de vectores con incrementos unitarios muy largos. Una sola instrucción podía realizar 65,000 cálculos usando tres entubamientos de memoria.

Caches Administradas por Software

He aquí una idea interesante: si un procesador vectorial puede planificar el arranque de un entubamiento de memoria con suficiente antelación, ¿por qué no puede un procesador RISC comenzar un llenado de cache antes de que requiera los datos en esa misma situación? De esta forma, está aprestando la cache para ocultar la latencia del llenado de la misma. Si puede realizarse con suficiente antelación, dará la impresión que todas las referencias a memoria operan a la velocidad de la cache.

Este concepto se llama precarga (prefetch) y está soportado mediante el uso de una instrucción especial de precarga disponible en muchos procesadores RISC. Tal instrucción opera exactamente igual que una operación de carga común, excepto que el procesador no espera que la cache se llene antes de completar la instrucción. La idea es precargar lo suficiente como para tener los datos listos en cache al momento en que se realiza el cálculo. El siguiente ejemplo ilustra cómo se usa:

```
DO I=1,1000000,8
  PREFETCH(ARR(I+8))
  DO J=0,7
    SUM=SUM+ARR(I+J)
  END DO
```

```
      END DO
```

Esto no es realmente FORTRAN. La precarga usualmente se realiza en el código ensamblador generado por el compilador cuando detecta que usted está saltando a lo largo del arreglo usando un salto de tamaño fijo. El compilador típicamente estima cuán adelante debe precargar. En el ejemplo anterior, si el llenado de la cache fuera particularmente lento, el valor 8 en I+8 puede cambiarse a 16 o 32 mientras los demás valores se cambien en consonancia.

En un procesador que sólo puede ejecutar una instrucción por ciclo, podría no valer la pena precargar una instrucción; el intercambio tomaría tiempo valioso en el flujo de instrucciones de entrada, con beneficios dudosos. En un procesador superescalar, sin embargo, puede mezclarse un aviso de cache con el resto del flujo de instrucciones, y emitirlo entre otras instrucciones reales. Si ello salva a su programa de sufrir fallos de cache extra, valdrá la pena.

Efectos Post-RISC sobre las Referencias a Memoria

En un procesador RISC, las operaciones de memoria típicamente acceden a ésta durante la fase de ejecución del entubamiento. En el procesador post-RISC, las cosas no son diferentes que en un procesador RISC, excepto que en un momento dado muchas cargas pueden estar a medio terminar. En algunos procesadores actuales pueden estar activas hasta 28 operaciones de memoria, con 10 en espera de salir de la memoria. Esta es una forma excelente de compensar la latencia de una memoria lenta comparada con la velocidad de la CPU. Considere el siguiente ciclo:

```
             LOADI     R6,10000    El número de
iteraciones
             LOADI     R5,0        La variable
índice
     CICLO:  LOAD      R1,R2(R5)   Carga un valor
```

```
de la memoria
            INCR      R1          Suma 1 a R1
            STORE     R1,R3(R5)   Almacena el
valor incrementado de vuelta en la memoria
            INCR      R5          Suma 1 a R5
            COMPARE   R5,R6       Comprueba la
terminación del ciclo
            BLT       CICLO       si R5 = R6,
salta a CICLO
```

Asumamos en este ejemplo que toma 50 ciclos acceder a la memoria. Cuando la búsqueda/decodificación pone la primera carga en el buffer de reordenamiento de instrucciones (IRB), la carga comienza en el siguiente ciclo y luego se suspende en la fase de ejecución. Sin embargo, el resto de las instrucciones están en el IRB. La instrucción INCR R1 debe esperar a la carga, y la instrucción STORE también debe esperar. Sin embargo, al usar un registro renombrado, las instrucciones INCR R5, COMPARE y BLT pueden todas ejecutarse, y la búsqueda/decodificación avanza a la parte superior del ciclo y envía otra carga al IRB para la siguiente localidad de memoria que tiene que esperar. Esta secuencia continúa hasta que se hayan cargado unas 10 iteraciones del ciclo en la TLB. Es entonces cuando se entrega la primera carga desde la memoria, y comienzan a ejecutarse las instrucciones INCR 1 y STORE de la primera iteración. Por supuesto el almacenamiento toma algo de tiempo, pero alrededor de ese momento termina la segunda carga, de modo que hay más trabajo por hacer, y así sucesivamente...

Como muchos otros aspectos de la computación, la arquitectura post-RISC, con su ejecución especulativa fuera de orden, optimiza las referencias a memoria. El procesador post-RISC desenrolla los ciclos dinámicamente, a tiempo de ejecución, para compensar los retrasos del subsistema de memoria. Asumiendo un sistema de memoria multibanco con entubamiento, que puede tener iniciadas múltiples operaciones de memoria antes de que alguna se complete (la HP PA-8000 tiene simultáneamente 10 operaciones de memoria fuera del chip al vuelo), el procesador continúa despachando operaciones a memoria hasta que tales operaciones comiencen a completarse.

A diferencia de un procesador vectorial o una instrucción de precarga, el procesador post-RISC no necesita anticipar el patrón preciso de referencias a memoria, de forma que puede controlar cuidadosamente el subsistema de memoria. Como resultado, el procesador post-RISC puede lograr su rendimiento máximo en un rango mucho más amplio de secuencias de código que los procesadores vectoriales y los procesadores RISC de ejecución en orden con capacidad de precarga.

La tolerancia implícita a la latencia de memoria hace a los procesadores post-RISC ideales para usarse en los procesadores escalables de memoria compartida del futuro, donde la jerarquía de memoria se hará todavía más compleja que en los procesadores actuales con tres niveles de cache y una memoria principal.

Desafortunadamente, el único segmento de código que no se beneficia significativamente de la arquitectura del post-RISC es el recorrido de listas ligadas. Ello se debe a que nunca se conoce la siguiente dirección hasta que se completa la carga previa, de forma que todas las cargas están fundamentalmente seriadas.

Tendencias en Tecnologías de RAM Dinámica

Muchas de las técnicas en esta sección se han enfocado en cómo lidiar con las imperfecciones de los chips de RAM dinámica (aunque cuando su tasa de velocidad de reloj alcanza los 300-600 MHz o 3-2 ns, incluso la SRAM comienza a parecer muy lenta). Es claro que la demanda por cada vez más RAM continuará incrementándose, y que digabits y más DRAM cupirán en un solo chip. Por ello, se está trabajando mucho para crear nuevas súper DRAMs más rápidas y mejor preparadas para los procesadores extremadamente rápidos del presente y del futuro. Algunas de las tecnologías son relativamente sencillas, mientras que otras requieren importantes rediseños en la forma en que procesadores y memorias se fabrican.

Entre las mejoras de la DRAM se incluyen:

- DRAM de modo de página rápida

- RAM de salida de datos extendida (EDO RAM)
- DRAM síncrona (SDRAM)
- RAMBUS
- DRAM con cache (CDRAM)

La DRAM de *modo de página* rápida ahorra tiempo al permitir un modo en el cuál no tiene que re-programarse la dirección completa en el chip para cada operación de memoria. En vez de ello, se asume que se accederá a la memoria secuencialmente (como en el llenado de una línea de cache), y sólo los bits de orden bajo de la dirección se modifican en las lecturas y escrituras sucesivas.

La *EDO RAM* es una modificación al mecanismo de buffer de salida en la RAM de modo de página, que le permite operar cerca del doble de rápido en operaciones que no sean de refresco.

La *DRAM síncrona* se sincroniza usando un reloj externo que permite a la cache y a la DRAM coordinar sus operaciones. Así, la SRAM puede entubar el recuperación de múltiples bits de memoria para mejorar el rendimiento global.

RAMBUS es una tecnología propietaria, capaz de transferir datos a 500 MB/seg. Usa una cantidad significativa de lógica adentro del chip, y opera a niveles de energía mayores que la DRAM típica.

La *DRAM con cache* combina una cache SRAM en el mismo chip que la DRAM. Con ello ambas quedan fuertemente acopladas, proporcionando rendimientos similares a dispositivos SRAM con todas las limitaciones de cualquier arquitectura de cache. Una ventaja del enfoque CDRAM es que incrementa la cantidad de cache y disminuye la cantidad de DRAM. También cuando se trabaja con sistemas de memoria con un gran número de intercalaciones, cada una tiene su propia SRAM para reducir la latencia, asumiendo que los datos solicitados estuvieran en la SRAM.

Un enfoque todavía más avanzado consiste en integrar procesador, SRAM y DRAM en un un único chip con un reloj a, digamos, 5 GHz, conteniendo 128 MB de datos. Comprensiblemente, hay una amplia variedad de problemas técnicos que resolver antes de que este tipo de componente esté

ampliamente disponible por US$200 -pero esa no es toda la cuestión. Los procesos de manufactura de la DRAM y los procesadores ya están comenzando a converger en algunas formas (RAMBUS). El mayor problema de rendimiento cuando tengamos esta clase de sistema será, "¿qué hacer si usted necesita 160 MB?"

Notas de Cierre

Se dice que la computadora del futuro será un buen sistema de memoria, que simplemente traiga conectada una CPU. Para que los sistemas de microprocesadores de alto rendimiento tomen el relevo como *los* motores de cómputo de alto rendimiento, el problema de un sistema de memoria basado en cache que usa DRAM para su memoria principal debe solucionarse. Hay muchos esfuerzos en desarrollo de arquitecturas y tecnologías, para transformar las memorias de las estaciones de trabajo y computadoras personales, y darles capacidades como las memorias de las supercomputadoras.

Conforme la velocidad de la CPU se incremente más rápido que la de la memoria, usted necesitará de las técnicas de este libro. Además, conforme se mueva hacia los procesadores múltiples, los problemas no mejorarán; usualmente se volverán peores. Con muchos procesadores constantemente hambrientos por más datos, un subsistema de memoria puede volverse extremadamente tenso.

Con sólo un poco de habilidad, a menudo podemos reestructurar los accesos a memoria, de forma tal que aprovechen las fortalezas de su sistema de memoria, en vez de sus debilidades.

Ejercicios

Exercise:

Problem:

El siguiente segmento de código recorre una cadena de apuntadores: `while ((p = (char *) *p) != NULL);` ¿Cómo interactuará este código con la cache si todas las referencias caen en una pequeña porción de memoria? ¿Cómo interactuará con la cache si las referencias se prolongan a través de varios megabytes?

Exercise:

Problem:

¿Cómo se comportará el código en [link] en un sistema de memoria multibanco que no tenga cache?

Exercise:

Problem:

Hace mucho tiempo, la gente escribía regularmente código automodificable -programas que escriben en la memoria de instrucciones y cambian su propio comportamiento. ¿Cuáles son las implicaciones de código automodificable en una máquina con una arquitectura de memoria Harvard?

Exercise:

Problem:

Asuma una arquitectura de memoria con una velocidad de cache L1 de 10 ns, una velocidad L2 de 30 ns, y una velocidad de memoria de 200 ns. Compare el rendimiento promedio del sistema de memoria con: 1) L1 80%, L2 10% y memoria 10%; y 2) L1 85% y memoria 15%.

Exercise:

Problem:

En un sistema de cómputo, ejecute ciclos que procesen arreglos de longitudes variables de 16 a 16 millones: `ARRAY(I) = ARRAY(I) + 3` ¿Cómo cambia el número de sumas por segundo conforme la longitud del arreglo cambia? Experimente con `REAL*4`, `REAL*8`, `INTEGER*4`, e `INTEGER*8`.

¿Qué tiene un impacto más significativo en el rendimiento, elementos de arreglo más grandes o números enteros versus números de punto flotante? Realice pruebas en una variedad de computadoras diferentes.

Exercise:

Problem:

Elabore un arreglo bidimensional de 1024x1024. Recorra el array, con los renglones como ciclo interno y después con las columnas como ciclo interno. Realice alguna operación simple en cada elemento. ¿Se comportan distinto los ciclos? ¿Por qué? Experimente con diferentes dimensiones de arreglos y vea el impacto en el rendimiento.

Exercise:

Problem:

Escriba un programa que ejecute repetidamente ciclos temporizados de diferentes tamaños, para determinar el tamaño de cache de su sistema.

Introduction

Often when we want to make a point that nothing is sacred, we say, “one plus one does not equal two.” This is designed to shock us and attack our fundamental assumptions about the nature of the universe. Well, in this chapter on floating- point numbers, we will learn that “ does not always equal ” when we use floating-point numbers for computations.

In this chapter we explore the limitations of floating-point numbers and how you as a programmer can write code to minimize the effect of these limitations. This chapter is just a brief introduction to a significant field of mathematics called *numerical analysis*.

Reality

The real world is full of real numbers. Quantities such as distances, velocities, masses, angles, and other quantities are all real numbers. [footnote] A wonderful property of real numbers is that they have unlimited accuracy. For example, when considering the ratio of the circumference of a circle to its diameter, we arrive at a value of 3.141592.... The decimal value for *pi* does not terminate. Because real numbers have unlimited accuracy, even though we can't write it down, *pi* is still a real number. Some real numbers are rational numbers because they can be represented as the ratio of two integers, such as 1/3. Not all real numbers are rational numbers. Not surprisingly, those real numbers that aren't rational numbers are called irrational. You probably would not want to start an argument with an irrational number unless you have a lot of free time on your hands.
In high performance computing we often simulate the real world, so it is somewhat ironic that we use simulated real numbers (floating-point) in those simulations of the real world.

Unfortunately, on a piece of paper, or in a computer, we don't have enough space to keep writing the digits of *pi*. So what do we do? We decide that we only need so much accuracy and round real numbers to a certain number of digits. For example, if we decide on four digits of accuracy, our approximation of *pi* is 3.142. Some state legislature attempted to pass a law that *pi* was to be three. While this is often cited as evidence for the IQ of governmental entities, perhaps the legislature was just suggesting that we only need one digit of accuracy for *pi*. Perhaps they foresaw the need to save precious memory space on computers when representing real numbers.

Representation

Given that we cannot perfectly represent real numbers on digital computers, we must come up with a compromise that allows us to approximate real numbers.[footnote] There are a number of different ways that have been used to represent real numbers. The challenge in selecting a representation is the trade-off between space and accuracy and the tradeoff between speed and accuracy. In the field of high performance computing we generally expect our processors to produce a floating- point result every 600-MHz clock cycle. It is pretty clear that in most applications we aren't willing to drop this by a factor of 100 just for a little more accuracy. Before we discuss the format used by most high performance computers, we discuss some alternative (albeit slower) techniques for representing real numbers. Interestingly, analog computers have an easier time representing real numbers. Imagine a "water- adding" analog computer which consists of two glasses of water and an empty glass. The amount of water in the two glasses are perfectly represented real numbers. By pouring the two glasses into a third, we are adding the two real numbers perfectly (unless we spill some), and we wind up with a real number amount of water in the third glass. The problem with analog computers is knowing just how much water is in the glasses when we are all done. It is also problematic to perform 600 million additions per second using this technique without getting pretty wet. Try to resist the temptation to start an argument over whether quantum mechanics would cause the real numbers to be rational numbers. And don't point out the fact that even digital computers are really analog computers at their core. I am trying to keep the focus on floating-point values, and you keep drifting away!

Binary Coded Decimal

In the earliest computers, one technique was to use binary coded decimal (BCD). In BCD, each base-10 digit was stored in four bits. Numbers could be arbitrarily long with as much precision as there was memory:

```
123.45
0001 0010 0011 0100 0101
```

This format allows the programmer to choose the precision required for each variable. Unfortunately, it is difficult to build extremely high-speed hardware to perform arithmetic operations on these numbers. Because each number may be far longer than 32 or 64 bits, they did not fit nicely in a register. Much of the floating- point operations for BCD were done using loops in microcode. Even with the flexibility of accuracy on BCD representation, there was still a need to round real numbers to fit into a limited amount of space.

Another limitation of the BCD approach is that we store a value from 0–9 in a four-bit field. This field is capable of storing values from 0–15 so some of the space is wasted.

Rational Numbers

One intriguing method of storing real numbers is to store them as rational numbers. To briefly review mathematics, rational numbers are the subset of real numbers that can be expressed as a ratio of integer numbers. For example, 22/7 and 1/2 are rational numbers. Some rational numbers, such as 1/2 and 1/10, have perfect representation as base-10 decimals, and others, such as 1/3 and 22/7, can only be expressed as infinite-length base-10 decimals. When using rational numbers, each real number is stored as two integer numbers representing the numerator and denominator. The basic fractional arithmetic operations are used for addition, subtraction, multiplication, and division, as shown in [link].

Rational number mathematics

$$\frac{1}{3} \times \frac{30}{7} = \frac{30}{21} = \frac{10}{7}$$

$$\frac{1}{6} + \frac{1}{5} = \frac{5}{30} + \frac{6}{30} = \frac{11}{30}$$

$$\frac{14173}{21224} \times \frac{77234}{2121} = \frac{1094637482}{45016104} = \frac{547318741}{22508052}$$

The limitation that occurs when using rational numbers to represent real numbers is that the size of the numerators and denominators tends to grow. For each addition, a common denominator must be found. To keep the numbers from becoming extremely large, during each operation, it is important to find the *greatest common divisor* (GCD) to reduce fractions to their most compact representation. When the values grow and there are no common divisors, either the large integer values must be stored using dynamic memory or some form of approximation must be used, thus losing the primary advantage of rational numbers.

For mathematical packages such as Maple or Mathematica that need to produce exact results on smaller data sets, the use of rational numbers to represent real numbers is at times a useful technique. The performance and storage cost is less significant than the need to produce exact results in some instances.

Fixed Point

If the desired number of decimal places is known in advance, it's possible to use fixed-point representation. Using this technique, each real number is stored as a scaled integer. This solves the problem that base-10 fractions such as 0.1 or 0.01 cannot be perfectly represented as a base-2 fraction. If you multiply 110.77 by 100 and store it as a scaled integer 11077, you can perfectly represent the base-10 fractional part (0.77). This approach can be used for values such as money, where the number of digits past the decimal point is small and known.

However, just because all numbers can be accurately represented it doesn't mean there are not errors with this format. When multiplying a fixed-point number by a fraction, you get digits that can't be represented in a fixed-point format, so some form of rounding must be used. For example, if you have $125.87 in the bank at 4% interest, your interest amount would be $5.0348. However, because your bank balance only has two digits of accuracy, they only give you $5.03, resulting in a balance of $130.90. Of course you probably have heard many stories of programmers getting rich depositing many of the remaining 0.0048 amounts into their own account. My guess is that banks have probably figured that one out by now, and the

bank keeps the money for itself. But it does make one wonder if they round or truncate in this type of calculation.[footnote]
Perhaps banks round this instead of truncating, knowing that they will always make it up in teller machine fees.

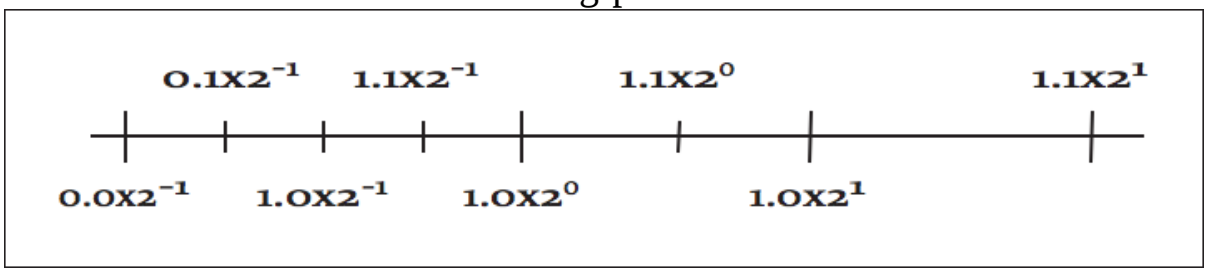
Mantissa/Exponent

The floating-point format that is most prevalent in high performance computing is a variation on scientific notation. In scientific notation the real number is represented using a mantissa, base, and exponent: 6.02×10^{23}.

The mantissa typically has some fixed number of places of accuracy. The mantissa can be represented in base 2, base 16, or BCD. There is generally a limited range of exponents, and the exponent can be expressed as a power of 2, 10, or 16.

The primary advantage of this representation is that it provides a wide overall range of values while using a fixed-length storage representation. The primary limitation of this format is that the difference between two successive values is not uniform. For example, assume that you can represent three base-10 digits, and your exponent can range from –10 to 10. For numbers close to zero, the "distance" between successive numbers is very small. For the number 1.72×10^{-10}, the next larger number is 1.73×10^{-10}. The distance between these two "close" small numbers is 0.000000000001. For the number 6.33×10^{10}, the next larger number is 6.34×10^{10}. The distance between these "close" large numbers is 100 million.

In [link], we use two base-2 digits with an exponent ranging from –1 to 1.
Distance between successive floating-point numbers

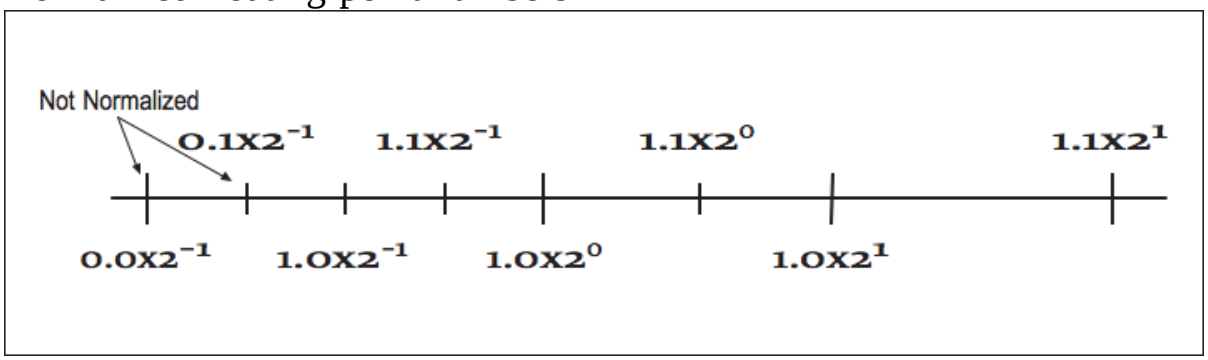


There are multiple equivalent representations of a number when using scientific notation:

6.00×10^5
0.60×10^6
0.06×10^7

By convention, we shift the mantissa (adjust the exponent) until there is exactly one nonzero digit to the left of the decimal point. When a number is expressed this way, it is said to be "normalized." In the above list, only 6.00×10^5 is normalized. [link] shows how some of the floating-point numbers from [link] are not normalized.

While the mantissa/exponent has been the dominant floating-point approach for high performance computing, there were a wide variety of specific formats in use by computer vendors. Historically, each computer vendor had their own particular format for floating-point numbers. Because of this, a program executed on several different brands of computer would generally produce different answers. This invariably led to heated discussions about which system provided the right answer and which system(s) were generating meaningless results.[footnote]
Interestingly, there was an easy answer to the question for many programmers. Generally they trusted the results from the computer they used to debug the code and dismissed the results from other computers as garbage.
Normalized floating-point numbers



When storing floating-point numbers in digital computers, typically the mantissa is normalized, and then the mantissa and exponent are converted to base-2 and packed into a 32- or 64-bit word. If more bits were allocated

to the exponent, the overall range of the format would be increased, and the number of digits of accuracy would be decreased. Also the base of the exponent could be base-2 or base-16. Using 16 as the base for the exponent increases the overall range of exponents, but because normalization must occur on four-bit boundaries, the available digits of accuracy are reduced on the average. Later we will see how the IEEE 754 standard for floating-point format represents numbers.

Effects of Floating-Point Representation

One problem with the mantissa/base/exponent representation is that not all base-10 numbers can be expressed perfectly as a base-2 number. For example, 1/2 and 0.25 can be represented perfectly as base-2 values, while 1/3 and 0.1 produce infinitely repeating base-2 decimals. These values must be rounded to be stored in the floating-point format. With sufficient digits of precision, this generally is not a problem for computations. However, it does lead to some anomalies where algebraic rules do not appear to apply. Consider the following example:

```
REAL*4 X,Y
X = 0.1
Y = 0
DO I=1,10
  Y = Y + X
ENDDO
IF ( Y .EQ. 1.0 ) THEN
  PRINT *,'Algebra is truth'
ELSE
  PRINT *,'Not here'
ENDIF
PRINT *,1.0-Y
END
```

At first glance, this appears simple enough. Mathematics tells us ten times 0.1 should be one. Unfortunately, because 0.1 cannot be represented exactly as a base-2 decimal, it must be rounded. It ends up being rounded down to the last bit. When ten of these slightly smaller numbers are added together, it does not quite add up to 1.0. When X and Y are `REAL*4`, the difference is about 10^{-7}, and when they are `REAL*8`, the difference is about 10^{-16}.

One possible method for comparing computed values to constants is to subtract the values and test to see how close the two values become. For example, one can rewrite the test in the above code to be:

```
IF ( ABS(1.0-Y).LT. 1E-6) THEN
  PRINT *,'Close enough for government work'
ELSE
  PRINT *,'Not even close'
ENDIF
```

The type of the variables in question and the expected error in the computation that produces Y determines the appropriate value used to declare that two values are close enough to be declared equal.

Another area where inexact representation becomes a problem is the fact that algebraic inverses do not hold with all floating-point numbers. For example, using `REAL*4`, the value `(1.0/X) * X` does not evaluate to 1.0 for 135 values of X from one to 1000. This can be a problem when computing the inverse of a matrix using LU-decomposition. LU-decomposition repeatedly does division, multiplication, addition, and subtraction. If you do the straightforward LU-decomposition on a matrix with integer coefficients that has an integer solution, there is a pretty good chance you won't get the exact solution when you run your algorithm. Discussing techniques for improving the accuracy of matrix inverse computation is best left to a numerical analysis text.

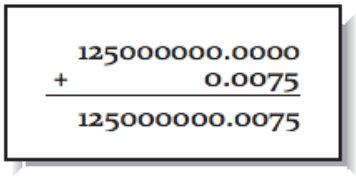
More Algebra That Doesn't Work

While the examples in the proceeding section focused on the limitations of multiplication and division, addition and subtraction are not, by any means, perfect. Because of the limitation of the number of digits of precision, certain additions or subtractions have no effect. Consider the following example using `REAL*4` with 7 digits of precision:

```
X = 1.25E8
Y = X + 7.5E-3
IF ( X.EQ.Y ) THEN
  PRINT *,'Am I nuts or what?'
ENDIF
```

While both of these numbers are precisely representable in floating-point, adding them is problematic. Prior to adding these numbers together, their decimal points must be aligned as in [link].

Figure 4-4: Loss of accuracy while aligning decimal points



Unfortunately, while we have computed the exact result, it cannot fit back into a `REAL*4` variable (7 digits of accuracy) without truncating the 0.0075. So after the addition, the value in Y is exactly 1.25E8. Even sadder, the addition could be performed *millions* of times, and the value for Y would still be 1.25E8.

Because of the limitation on precision, not all algebraic laws apply all the time. For instance, the answer you obtain from X+Y will be the same as Y+X, as per the commutative law for addition. Whichever operand you pick first, the operation yields the same result; they are mathematically

equivalent. It also means that you can choose either of the following two forms and get the same answer:

```
(X + Y) + Z
(Y + X) + Z
```

However, this is not equivalent:

```
(Y + Z) + X
```

The third version isn't equivalent to the first two because the order of the calculations has changed. Again, the rearrangement is equivalent algebraically, but not computationally. By changing the order of the calculations, we have taken advantage of the associativity of the operations; we have made an *associative transformation* of the original code.

To understand why the order of the calculations matters, imagine that your computer can perform arithmetic significant to only five decimal places.

Also assume that the values of X, Y, and Z are .00005, .00005, and 1.0000, respectively. This means that:

```
(X + Y) + Z = .00005 + .00005 + 1.0000
            = .0001           + 1.0000     =
1.0001
```

but:

```
(Y + Z) + X =  .00005 + 1.0000 + .00005
```

```
                    = 1.0000            + .00005   =
1.0000
```

The two versions give slightly different answers. When adding `Y+Z+X`, the sum of the smaller numbers was insignificant when added to the larger number. But when computing `X+Y+Z`, we add the two small numbers first, and their combined sum is large enough to influence the final answer. For this reason, compilers that rearrange operations for the sake of performance generally only do so after the user has requested optimizations beyond the defaults.

For these reasons, the FORTRAN language is very strict about the exact order of evaluation of expressions. To be compliant, the compiler must ensure that the operations occur exactly as you express them.[footnote] Often even if you didn't mean it.

For Kernighan and Ritchie C, the operator precedence rules are different. Although the precedences between operators are honored (i.e., * comes before +, and evaluation generally occurs left to right for operators of equal precedence), the compiler is allowed to treat a few commutative operations (+, *, &, ^ and |) as if they were fully associative, *even* if they are parenthesized. For instance, you might tell the C compiler:

```
a = x + (y + z);
```

However, the C compiler is free to ignore you, and combine X, Y, and Z in any order it pleases.

Now armed with this knowledge, view the following harmless-looking code segment:

```
      REAL*4 SUM,A(1000000)
      SUM = 0.0
      DO I=1,1000000
```

```
  SUM = SUM + A(I)
ENDDO
```

Begins to look like a nightmare waiting to happen. The accuracy of this sum depends of the relative magnitudes and order of the values in the array A. If we sort the array from smallest to largest and then perform the additions, we have a more accurate value. There are other algorithms for computing the sum of an array that reduce the error without requiring a full sort of the data. Consult a good textbook on numerical analysis for the details on these algorithms.

If the range of magnitudes of the values in the array is relatively small, the straight- forward computation of the sum is probably sufficient.

Improving Accuracy Using Guard Digits

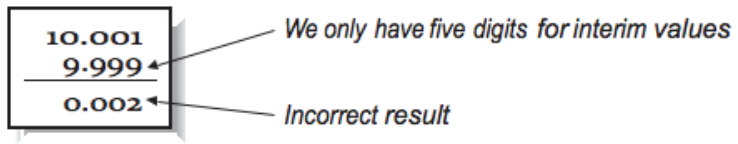
In this section we explore a technique to improve the precision of floating-point computations without using additional storage space for the floating-point numbers.

Consider the following example of a base-10 system with five digits of accuracy performing the following subtraction:

```
10.001 - 9.9993 = 0.0017
```

All of these values can be perfectly represented using our floating-point format. However, if we only have five digits of precision available while aligning the decimal points during the computation, the results end up with significant error as shown in [link].

Need for guard digits



To perform this computation and round it correctly, we do not need to increase the number of significant digits for *stored* values. We do, however, need additional digits of precision while performing the computation.

The solution is to add extra *guard digits* which are maintained during the interim steps of the computation. In our case, if we maintained six digits of accuracy while aligning operands, and rounded before normalizing and assigning the final value, we would get the proper result. The guard digits only need to be present as part of the floating-point execution unit in the CPU. It is not necessary to add guard digits to the registers or to the values stored in memory.

It is not necessary to have an extremely large number of guard digits. At some point, the difference in the magnitude between the operands becomes so great that lost digits do not affect the addition or rounding results.

History of IEEE Floating-Point Format

History of IEEE Floating-Point Format

Prior to the RISC microprocessor revolution, each vendor had their own floating-point formats based on their designers' views of the relative importance of range versus accuracy and speed versus accuracy. It was not uncommon for one vendor to carefully analyze the limitations of another vendor's floating-point format and use this information to convince users that theirs was the only "accurate" floating-point implementation. In reality none of the formats was perfect. The formats were simply imperfect in different ways.

During the 1980s the Institute for Electrical and Electronics Engineers (IEEE) produced a standard for the floating-point format. The title of the standard is "IEEE 754-1985 Standard for Binary Floating-Point Arithmetic." This standard provided the precise definition of a floating-point format and described the operations on floating-point values.

Because IEEE 754 was developed after a variety of floating-point formats had been in use for quite some time, the IEEE 754 working group had the benefit of examining the existing floating-point designs and taking the strong points, and avoiding the mistakes in existing designs. The IEEE 754 specification had its beginnings in the design of the Intel i8087 floating-point coprocessor. The i8087 floating-point format improved on the DEC VAX floating-point format by adding a number of significant features.

The near universal adoption of IEEE 754 floating-point format has occurred over a 10-year time period. The high performance computing vendors of the mid 1980s (Cray IBM, DEC, and Control Data) had their own proprietary floating-point formats that they had to continue supporting because of their installed user base. They really had no choice but to continue to support their existing formats. In the mid to late 1980s the primary systems that supported the IEEE format were RISC workstations and some coprocessors for microprocessors. Because the designers of these systems had no need to protect a proprietary floating-point format, they readily adopted the IEEE format. As RISC processors moved from general-purpose integer computing to high performance floating-point computing, the CPU designers found ways to make IEEE floating-point operations operate very quickly. In 10 years, the IEEE 754 has gone from a standard for floating-point coprocessors to the dominant floating-point standard for all computers. Because of this standard, we, the users, are the beneficiaries of a portable floating-point environment.

IEEE Floating-Point Standard

The IEEE 754 standard specified a number of different details of floating-point operations, including:

- Storage formats
- Precise specifications of the results of operations
- Special values
- Specified runtime behavior on illegal operations

Specifying the floating-point format to this level of detail insures that when a computer system is compliant with the standard, users can expect repeatable execution from one hardware platform to another when operations are executed in the same order.

IEEE Storage Format

The two most common IEEE floating-point formats in use are 32- and 64-bit numbers. [link] gives the general parameters of these data types.

IEEE75	FORTRAN	C	Bits	Exponent Bits	Mantissa Bits
Single	REAL*4	float	32	8	24
Double	REAL*8	double	64	11	53
Double-Extended	REAL*10	long double	>=80	>=15	>=64

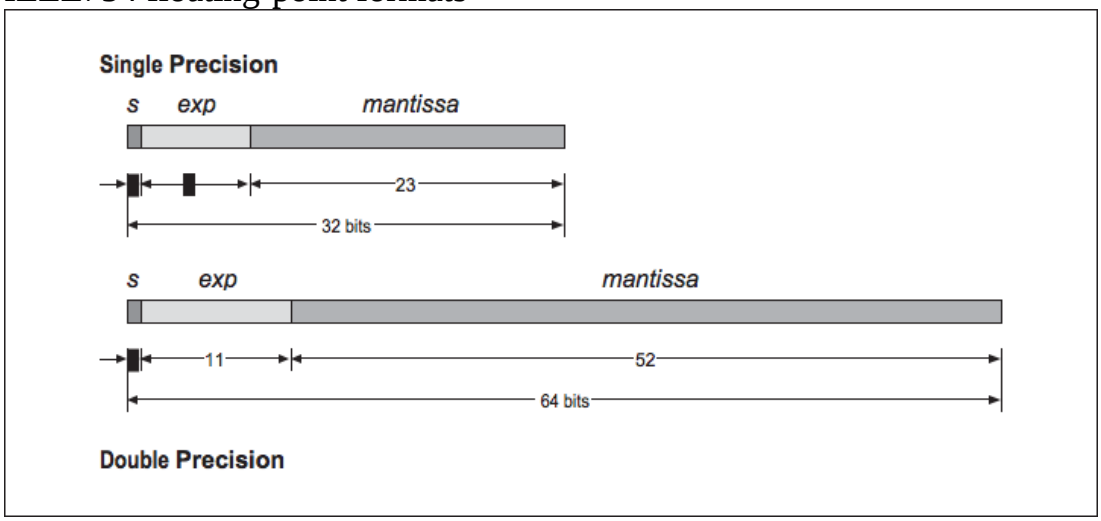
Parameters of IEEE 32- and 64-Bit Formats

In FORTRAN, the 32-bit format is usually called REAL, and the 64-bit format is usually called DOUBLE. However, some FORTRAN compilers double the sizes for these data types. For that reason, it is safest to declare your FORTRAN variables as `REAL*4` or `REAL*8`. The double-extended format is not as well

supported in compilers and hardware as the single- and double-precision formats. The bit arrangement for the single and double formats are shown in [link].

Based on the storage layouts in [link], we can derive the ranges and accuracy of these formats, as shown in [link].

IEEE754 floating-point formats



IEEE754	Minimum Normalized Number	Largest Finite Number	Base-10 Accuracy
Single	1.2E-38	3.4 E+38	6-9 digits
Double	2.2E-308	1.8 E+308	15-17 digits
Extended Double	3.4E-4932	1.2 E+4932	18-21 digits

Range and Accuracy of IEEE 32- and 64-Bit Formats

Converting from Base-10 to IEEE Internal Format

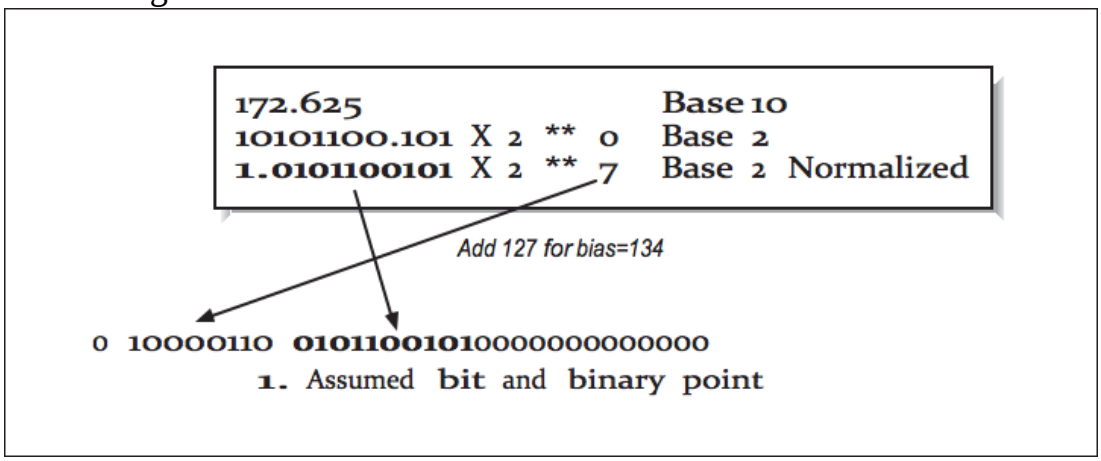
We now examine how a 32-bit floating-point number is stored. The high-order bit is the sign of the number. Numbers are stored in a sign-magnitude format (i.e., not 2's - complement). The exponent is stored in the 8-bit field biased by adding 127 to the exponent. This results in an exponent ranging from -126 through +127.

The mantissa is converted into base-2 and normalized so that there is one nonzero digit to the left of the binary place, adjusting the exponent as necessary. The digits to the right of the binary point are then stored in the low-order 23 bits of the word. Because all numbers are normalized, there is no need to store the leading 1.

This gives a free extra bit of precision. Because this bit is dropped, it's no longer proper to refer to the stored value as the mantissa. In IEEE parlance, this mantissa minus its leading digit is called the *significand*.

[link] shows an example conversion from base-10 to IEEE 32-bit format.

Converting from base-10 to IEEE 32-bit format



The 64-bit format is similar, except the exponent is 11 bits long, biased by adding 1023 to the exponent, and the significand is 54 bits long.

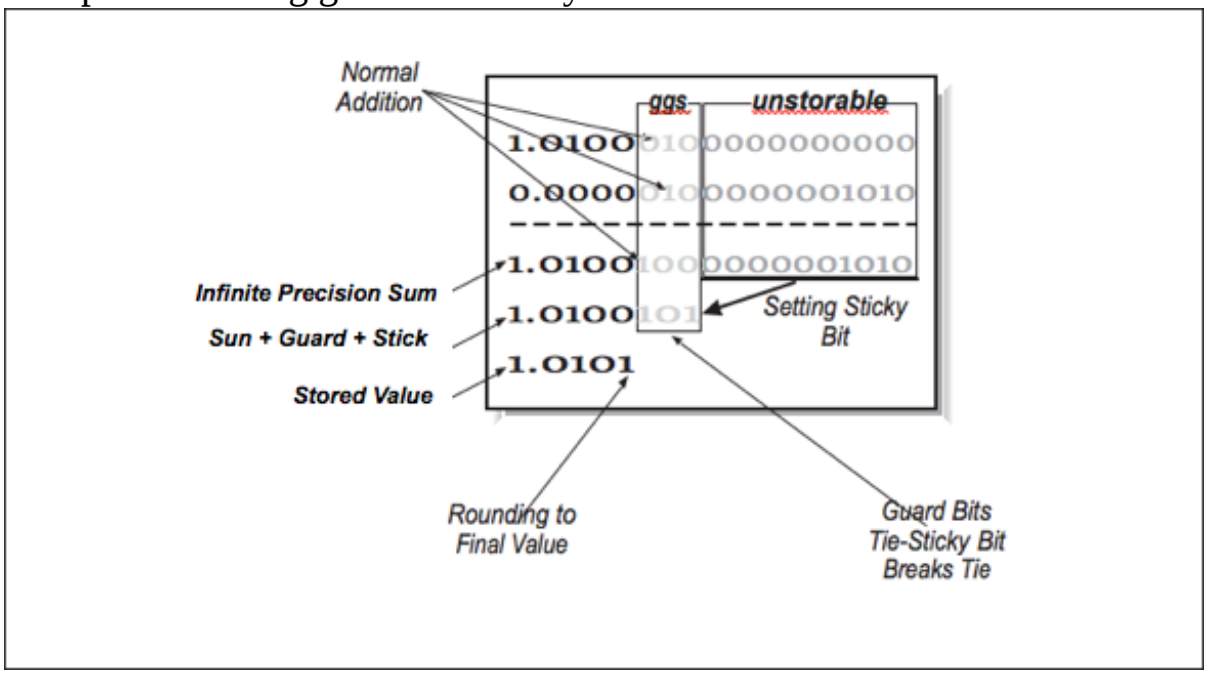
IEEE Operations

The IEEE standard specifies how computations are to be performed on floating- point values on the following operations:

- Addition
- Subtraction
- Multiplication
- Division
- Square root
- Remainder (modulo)
- Conversion to/from integer
- Conversion to/from printed base-10

These operations are specified in a machine-independent manner, giving flexibility to the CPU designers to implement the operations as efficiently as possible while maintaining compliance with the standard. During operations, the IEEE standard requires the maintenance of two guard digits and a sticky bit for intermediate values. The guard digits above and the sticky bit are used to indicate if any of the bits beyond the second guard digit is nonzero.

Computation using guard and sticky bits



In [link], we have five bits of normal precision, two guard digits, and a sticky bit. Guard bits simply operate as normal bits — as if the significand were 25 bits. Guard bits participate in rounding as the extended operands are added. The sticky bit is set to 1 if any of the bits beyond the guard bits is nonzero in either operand.[footnote] Once the extended sum is computed, it is rounded so that the value stored in memory is the closest possible value to the extended sum including the guard digits. [link] shows all eight possible values of the two guard digits and the sticky bit and the resulting stored value with an explanation as to why.

If you are somewhat hardware-inclined and you think about it for a moment, you will soon come up with a way to properly maintain the sticky bit without ever computing the full "infinite precision sum." You just have to keep track as things get shifted around.

Extended Sum	Stored Value	Why
1.0100 000	1.0100	Truncated based on guard digits
1.0100 001	1.0100	Truncated based on guard digits
1.0100 010	1.0100	Rounded down based on guard digits
1.0100 011	1.0100	Rounded down based on guard digits
1.0100 100	1.0100	Rounded down based on sticky bit
1.0100 101	1.0101	Rounded up based on sticky bit
1.0100 110	1.0101	Rounded up based on guard digits

1.0100 111	1.0101	Rounded up based on guard digits

Extended Sums and Their Stored Values

The first priority is to check the guard digits. Never forget that the sticky bit is just a hint, not a real digit. So if we can make a decision without looking at the sticky bit, that is good. The only decision we are making is to round the last storable bit up or down. When that stored value is retrieved for the next computation, its guard digits are set to zeros. It is sometimes helpful to think of the stored value as having the guard digits, but set to zero.

Two guard digits and the sticky bit in the IEEE format insures that operations yield the same rounding as if the intermediate result were computed using unlimited precision and then rounded to fit within the limits of precision of the final computed value.

At this point, you might be asking, “Why do I care about this minutiae?” At some level, unless you are a hardware designer, you don’t care. But when you examine details like this, you can be assured of one thing: when they developed the IEEE floating-point standard, they looked at the details *very* carefully. The goal was to produce the most accurate possible floating-point standard within the constraints of a fixed-length 32- or 64-bit format. Because they did such a good job, it’s one less thing you have to worry about. Besides, this stuff makes great exam questions.

Special Values

In addition to specifying the results of operations on numeric data, the IEEE standard also specifies the precise behavior on undefined operations such as dividing by zero. These results are indicated using several special values. These values are bit patterns that are stored in variables that are checked before operations are performed. The IEEE operations are all defined on these special values in addition to the normal numeric values. [link] summarizes the special values for a 32-bit IEEE floating-point number.

Special Value	Exponent	Significand
+ or – 0	00000000	0
Denormalized number	00000000	nonzero
NaN (Not a Number)	11111111	nonzero
+ or – Infinity	11111111	0

Special Values for an IEEE 32-Bit Number

The value of the exponent and significand determines which type of special value this particular floating-point number represents. Zero is designed such that integer zero and floating-point zero are the same bit pattern.

Denormalized numbers can occur at some point as a number continues to get smaller, and the exponent has reached the minimum value. We could declare that minimum to be the smallest representable value. However, with denormalized values, we can continue by setting the exponent bits to zero and shifting the significand bits to the right, first adding the leading “1” that was dropped, then continuing to add leading zeros to indicate even smaller values. At some point the last nonzero digit is shifted off to the right, and

the value becomes zero. This approach is called *gradual underflow* where the value keeps approaching zero and then eventually becomes zero. Not all implementations support denormalized numbers in hardware; they might trap to a software routine to handle these numbers at a significant performance cost.

At the top end of the biased exponent value, an exponent of all 1s can represent the *Not a Number* (NaN) value or infinity. Infinity occurs in computations roughly according to the principles of mathematics. If you continue to increase the magnitude of a number beyond the range of the floating-point format, once the range has been exceeded, the value becomes infinity. Once a value is infinity, further additions won't increase it, and subtractions won't decrease it. You can also produce the value infinity by dividing a nonzero value by zero. If you divide a nonzero value by infinity, you get zero as a result.

The NaN value indicates a number that is not mathematically defined. You can generate a NaN by dividing zero by zero, dividing infinity by infinity, or taking the square root of -1. The difference between infinity and NaN is that the NaN value has a nonzero significand. The NaN value is very sticky. Any operation that has a NaN as one of its inputs always produces a NaN result.

Exceptions and Traps

In addition to defining the results of computations that aren't mathematically defined, the IEEE standard provides programmers with the ability to detect when these special values are being produced. This way, programmers can write their code without adding extensive IF tests throughout the code checking for the magnitude of values. Instead they can register a trap handler for an event such as underflow and handle the event when it occurs. The exceptions defined by the IEEE standard include:

- Overflow to infinity
- Underflow to zero
- Division by zero
- Invalid operation
- Inexact operation

According to the standard, these traps are under the control of the user. In most cases, the compiler runtime library manages these traps under the direction from the user through compiler flags or runtime library calls. Traps generally have significant overhead compared to a single floating-point instruction, and if a program is continually executing trap code, it can significantly impact performance.

In some cases it's appropriate to ignore traps on certain operations. A commonly ignored trap is the underflow trap. In many iterative programs, it's quite natural for a value to keep reducing to the point where it "disappears." Depending on the application, this may or may not be an error situation so this exception can be safely ignored.

If you run a program and then it terminates, you see a message such as:

```
Overflow handler called 10,000,000 times
```

It probably means that you need to figure out why your code is exceeding the range of the floating-point format. It probably also means that your code is executing more slowly because it is spending too much time in its error handlers.

Compiler Issues

The IEEE 754 floating-point standard does a good job describing how floating- point operations are to be performed. However, we generally don't write assembly language programs. When we write in a higher-level language such as FORTRAN, it's sometimes difficult to get the compiler to generate the assembly language you need for your application. The problems fall into two categories:

- The compiler is too conservative in trying to generate IEEE-compliant code and produces code that doesn't operate at the peak speed of the processor. On some processors, to fully support gradual underflow, extra instructions must be generated for certain instructions. If your code will never underflow, these instructions are unnecessary overhead.
- The optimizer takes liberties rewriting your code to improve its performance, eliminating some necessary steps. For example, if you have the following code: `Z = X + 500 Y = Z - 200` The optimizer may replace it with `Y = X + 300`. However, in the case of a value for `X` that is close to overflow, the two sequences may not produce the same result.

Sometimes a user prefers "fast" code that loosely conforms to the IEEE standard, and at other times the user will be writing a numerical library routine and need total control over each floating-point operation. Compilers have a challenge supporting the needs of both of these types of users. Because of the nature of the high performance computing market and benchmarks, often the "fast and loose" approach prevails in many compilers.

Closing Notes

While this is a relatively long chapter with a lot of technical detail, it does not even begin to scratch the surface of the IEEE floating-point format or the entire field of numerical analysis. We as programmers must be careful about the accuracy of our programs, lest the results become meaningless. Here are a few basic rules to get you started:

- Look for compiler options that relax or enforce strict IEEE compliance and choose the appropriate option for your program. You may even want to change these options for different portions of your program.
- Use `REAL*8` for computations unless you are sure `REAL*4` has sufficient precision. Given that REAL*4 has roughly 7 digits of precision, if the bottom digits become meaningless due to rounding and computations, you are in some danger of seeing the effect of the errors in your results. `REAL*8` with 13 digits makes this much less likely to happen.
- Be aware of the relative magnitude of numbers when you are performing additions.
- When summing up numbers, if there is a wide range, sum from smallest to largest.
- Perform multiplications before divisions whenever possible.
- When performing a comparison with a computed value, check to see if the values are “close” rather than identical.
- Make sure that you are not performing any unnecessary type conversions during the critical portions of your code.

An excellent reference on floating-point issues and the IEEE format is “What Every Computer Scientist Should Know About Floating-Point Arithmetic,” written by David Goldberg, in ACM Computing Surveys magazine (March 1991). This article gives examples of the most common problems with floating-point and outlines the solutions. It also covers the IEEE floating-point format very thoroughly. I also recommend you consult Dr. William Kahan’s home page (http://www.cs.berkeley.edu/~wkahan/) for some excellent materials on the IEEE format and challenges using floating-point arithmetic. Dr. Kahan was one of the original designers of the Intel i8087 and the IEEE 754 floating-point format.

Exercises

Exercise:

Problem:

Run the following code to count the number of inverses that are not perfectly accurate:

```
REAL*4 X,Y,Z
INTEGER I
I = 0
DO X=1.0,1000.0,1.0
  Y = 1.0 / X
  Z = Y * X
  IF ( Z .NE. 1.0 ) THEN
    I = I + 1
  ENDIF
ENDDO
PRINT *,'Found ',I
END
```

Exercise:

Problem:

Change the type of the variables to `REAL*8` and repeat. Make sure to keep the optimization at a sufficiently low level (-00) to keep the compiler from eliminating the computations.

Exercise:

Problem:

Write a program to determine the number of digits of precision for `REAL*4` and `REAL*8`.

Exercise:

Problem:

Write a program to demonstrate how summing an array forward to backward and backward to forward can yield a different result.

Exercise:

Problem:

Assuming your compiler supports varying levels of IEEE compliance, take a significant computational code and test its overall performance under the various IEEE compliance options. Do the results of the program change?

Qué Hace un Compilador - Introducción

Lo que Hace un Compilador

El objetivo de un *compilador optimizador* es la traducción eficiente de un lenguaje de alto nivel al lenguaje máquina más rápido posible, que represente con precisión al primero. Lo que hace buena a una representación es: da las respuestas correctas, y se ejecuta rápidamente.

Naturalmente, no importa cuán rápido se ejecute un programa si no produce las respuestas correctas.[footnote] Pero dada una expresión de un programa que se ejecuta correctamente, un compilador optimizador busca formas de acelerarla. En primera instancia usualmente significa simplificar el código, retirando instrucciones extrañas, y compartiendo los resultados intermedios entre sentencias. Algunas optimizaciones más avanzadas buscan reestructurar el programa, a veces incrementando el tamaño del código, aunque (con algo de fortuna) el número de instrucciones ejecutadas se reduzca.
Sin embargo, a menudo se troca precisión por velocidad.

Cuando llegue finalmente el momento de generar lenguaje máquina, el compilador debe saber acerca de los registros y las reglas para emitir instrucciones. Para lograr alto desempeño, requiere entender el costo de tales instrucciones y las latencias de los recursos de la máquina, tales como las filas de espera. Esto es especialmente cierto para aquellos procesadores que pueden ejecutar más de una instrucción simultáneamente. Se necesita de una mezcla balanceada de instrucciones -la proporción correcta de operaciones de punto flotante, punto fijo, memoria, saltos, etc.- para mantener a la máquina ocupada.

Inicialmente los compiladores eran herramientas que nos permitían escribir en algo más legible que el lenguaje ensamblador. Hoy en día rondan la inteligencia artificial, conforme toman nuestro código fuente de alto nivel y lo traducen a una versión altamente optimizada de lenguaje máquina, a través de una amplia variedad de arquitecturas uniprocesador y multiprocesador. En el área del cómputo de alto rendimiento, a menudo sucede que el compilador tiene un impacto mayor sobre el desempeño de

nuestro programa que las arquitecturas del procesador o de la memoria. A lo largo de la historia del cómputo de alto rendimiento ha sucedido que, si no estábamos satisfechos con el desempeño de nuestro programa escrito en un lenguaje de alto nivel, alegremente reescribíamos todo o parte del mismo en lenguaje ensamblador. Afortunadamente, los compiladores actuales usualmente hacen este paso innecesario.

En este capítulo cubriremos la operación básica de los compiladores optimizadores. En un capítulo posterior cubriremos las técnicas usadas para analizar y compilar programas para arquitecturas avanzadas, tales como los sistemas paralelos o de procesamiento vectorial. Iniciaremos nuestra revisión de los compiladores, examinando cómo ha ido cambiando a lo largo del tiempo la relación entre los programadores y sus compiladores.

Qué Hace un Compilador - Historia de los Compiladores

Si usted ha formado parte del mundo del cómputo de alto rendimiento desde sus inicios en la década de 1950, le ha tocado programar en varios lenguajes desde entonces. Durante la década de 1950 e inicios de la de 1960, lo hizo en lenguaje ensamblador. La escasez de memoria y las bajas velocidades de reloj hacían que cada instrucción fuera preciosa. Con pequeñas memorias, el tamaño de los programas era típicamente pequeño, así que con el lenguaje ensamblador era suficiente. Para finales de la década de 1960, los programadores comenzaron a escribir más código en un lenguaje de alto nivel como FORTRAN. Usar uno de tales lenguajes hace que el trabajo que usted realice sea más transportable, confiable y fácil de mantener. Dado el incremento en velocidad y capacidad de las computadoras, el costo de usar un lenguaje de alto nivel fue algo que la mayoría de los programadores estaban dispuestos a aceptar. En la década de 1970, si un programa gastaba una cantidad particularmente grande de tiempo en cierta rutina, o la rutina formaba parte del sistema operativo, o se trataba de una biblioteca de uso común, muy probablemente estuviera escrita en ensamblador.

Durante la última parte de la década de 1970 e inicios de la de 1980, los *compiladores optimizadores* continuaron mejorando hasta el punto en que el grueso de los programas de propósito general, excepto las porciones más críticas, se escribían en lenguajes de alto nivel. En promedio, los compiladores generan mejor código que la mayoría de los programadores humanos de ensamblador. A menudo ello se debe a que el compilador puede hacer un mejor uso de algunos recursos de hardware, tales como los registros. En un procesador con 16 registros, un programador debe adoptar alguna convención respecto a qué registros usar para cada cosa, con el fin de poder seguirle la pista al valor que cada uno almacena. Un compilador puede usar cada registro como le plazca, porque puede darle un seguimiento preciso al momento en que está disponible para otro uso.

Sin embargo, durante ese periodo, también evolucionó la arquitectura de las computadoras de alto rendimiento. Cray Research estaba desarrollando procesadores vectoriales en el extremo superior del espectro computacional. Los compiladores todavía no estaban listos para determinar cuándo debían

emplear esas nuevas instrucciones vectoriales. Los programadores se vieron forzados a escribir lenguaje ensamblador o crear código FORTRAN muy afinado que invocaba a las rutinas vectoriales apropiadas en su código. En cierto sentido, los procesadores vectoriales hicieron girar las manecillas del reloj en el sentido inverso, hasta que comenzaron a confiar en el compilador. Los programadores nunca regresaron completamente al lenguaje ensamblador, pero su código FORTRAN comenzaba a lucir más como no-FORTRAN. Conforme maduraron las computadoras vectoriales, sus compiladores fueron progresivamente capaces de detectar cuándo podía realizarse la vectorización. Y en algún momento nuevamente se hicieron mejores que los programadores humanos en tales arquitecturas. Los nuevos compiladores reducían la necesidad de usar intensivamente directivas o extensiones del lenguaje.[footnote]
Los Ciclos de Livermore eran una prueba comparativa que específicamente comprobaba la capacidad del compilador para optimizar eficientemente un conjunto de ciclos. Además de ser una prueba comparativa de rendimiento, también comparaba los compiladores.

La revolución RISC descansaba en una dependencia creciente respecto al compilador. Programar en los primeros procesadores RISC, tales como el Intel i860 resultaba doloroso si se comparaba con los procesadores CISC. Sutiles diferencias en la forma de codificar un programa en lenguaje máquina podían tener un impacto significativo en el rendimiento global del programa. Por ejemplo, un programador debía contar los ciclos transcurridos entre una instrucción de carga y el uso de los resultados de la carga en una instrucción posterior de cálculo. Conforme se desarrollaron procesadores superescalares, se hizo necesario volver simultáneas ciertos pares de instrucciones, y otras debían acomodarse secuencialmente. Dado que había en el mercado un gran número de procesadores RISC distintos, los programadores no tenían tiempo de aprender tales particularidades para lograr hasta la última gota de rendimiento en cada procesador. Era más fácil mantener unidos al diseñador del procesador y el escritor del compilador (que con suerte trabajaban para la misma compañía) y hacer que discutieran a fondo la mejor forma de generar el código máquina. Y entonces cualquiera podía usar el compilador y obtener código que hiciera un uso razonablemente bueno del hardware.

El compilador se convirtió en una herramienta importante en el ciclo de diseño del procesador. Los diseñadores de procesadores tenían mucha mayor flexibilidad respecto a los tipos de cambios que podían hacer. Por ejemplo, podía suceder que un buen diseño en la siguiente revisión de un procesador ejecutase código previo 10% más lento que una revisión nueva, pero recompilando el código lo hiciera 65% más rápido. Por supuesto, era importante proporcionar dicho compilador con cada nuevo embarque de procesadores, y que dicho compilador diera ese nivel de rendimiento en una amplia variedad de código, y no sólo en una suite de pruebas de escritorio en particular.

Qué Hace un Compilador - Cuál Lenguaje Optimizar

Se ha dicho, "No sé qué lenguaje se usará para programar computadoras de alto rendimiento dentro de 10 años, pero sí que se llamará FORTRAN". A riesgo de iniciar una guerra de exterminio, necesitamos discutir las fortalezas y debilidades de los lenguajes que se usan para cómputo de alto rendimiento. Muchos científicos de computadoras (no científicos de la computación) se han entrenado en una dieta continua de C, C++[footnote] o algún otro lenguaje enfocado en estructuras de datos u objetos. Cuando los estudiantes se topan por vez primera con el cómputo de alto rendimiento, tienen el deseo instintivo de continuar programando en su lenguaje favorito. Sin embargo, si se quiere obtener el rendimiento máximo en un extenso rango de arquitecturas, FORTRAN es el único lenguaje práctico.
Que conste que ambos autores de este libro son consumados expertos en C, C++ y FORTRAN, y no tienen nociones preconcebidas.

Cuando los estudiantes preguntan el porqué, usualmente la primera respuesta es "porque siempre ha sido así". Y en cierta forma es cierto. Físicos, ingenieros mecánicos, químicos, ingenieros estructurales y meteorólogos realizan mucha programación en computadoras de alto rendimiento, y FORTRAN es el lenguaje en esos campos. (¿Cuándo fue la última vez que un estudiante de ciencias de la computación escribió un programa que funcione apropiadamente, y que calcule durante una semana completa?) Así que de forma natural los vendedores de computadoras de alto rendimiento ponen un gran esfuerzo en hacer que FORTRAN trabaje muy bien en la arquitectura que comercializan.

Pero, claro está, ésta no es la única razón por la cual FORTRAN es el mejor lenguaje. Hay algunos elementos fundamentales que hacen que C, C++ o cualesquiera otro lenguaje orientado a las estructuras de datos no resulte adecuado para la programación de alto rendimiento. En una palabra, el problema son los *apuntadores*. Los apuntadores (o direcciones) son la forma mediante la cual los buenos científicos de la computación construyen listas ligadas, árboles binarios, colas dobles y toda clase de elegantes estructuras de datos. El problema estriba en que el efecto de una operación con apuntadores sólo se conoce a tiempo de ejecución, cuando el valor de dicho apuntador se carga en memoria. Una vez que el optimizador del

compilador ve un apuntador, se vienen abajo todas las apuestas. No puede hacerse ninguna suposición sobre el efecto de una operación de apuntadores a tiempo de compilación. Debe generarse código conservador (menos optimizado) que simplemente hace exactamente la misma operación en lenguaje máquina, que la que describe el lenguaje de alto nivel.

Aunque la carencia de apuntadores en FORTRAN es una ventaja para la optimización, limita seriamente la habilidad del programador para crear estructuras de datos. En ciertas aplicaciones, especialmente en aquellas altamente escalables y basadas en redes, el uso de las estructuras de datos adecuadas puede mejorar significativamente el rendimiento global de la aplicación. Para resolverlo, en la especificación de FORTRAN 90 se han agregado apuntadores al lenguaje. En cierto modo, se trata de un intento de la comunidad FORTRAN para evitar que los programadores usen C en sus aplicaciones en aquellas áreas donde requieren del uso de estructuras de datos. Si los programadores comienzan a usar apuntadores en sus códigos, sus programas en FORTRAN sufrirán los mismos problemas que inhiben la optimización de los programas en C. En cierto modo FORTRAN ha renunciado a su principal ventaja respecto a C, por tratar de parecerse a C. El debate acerca de los apuntadores es una de las razones que ha frenado la tasa de adopción de FORTRAN 90. Muchos programadores prefieren crear sus estructuras de datos, comunicaciones y otro trabajo de contabilidad en C, y hacer sus cálculos en FORTRAN 77.

FORTRAN 90 también tiene fortalezas y debilidades cuando se le compara con FORTRAN 77 en plataformas de cómputo de alto rendimiento. FORTRAN 90 tiene una fuerte ventaja sobre FORTRAN 77 en su semántica mejorada, que da mayores oportunidades para optimizaciones avanzadas. Esta ventaja es especialmente cierta sobre sistemas de memoria distribuida, en los que la descomposición de datos es un factor significativo. (Véase [link].) Sin embargo, hasta que FORTRAN 90 se vuelva popular, los vendedores no tendrán motivación para exprimir hasta el último bit de rendimiento de FORTRAN 90.

Así que mientras FORTRAN 77 continúe siendo el lenguaje principal para el cómputo de alto rendimiento en el futuro cercano, otros lenguajes como C y FORTRAN 90 tendrán que jugar papeles limitados y potencialmente

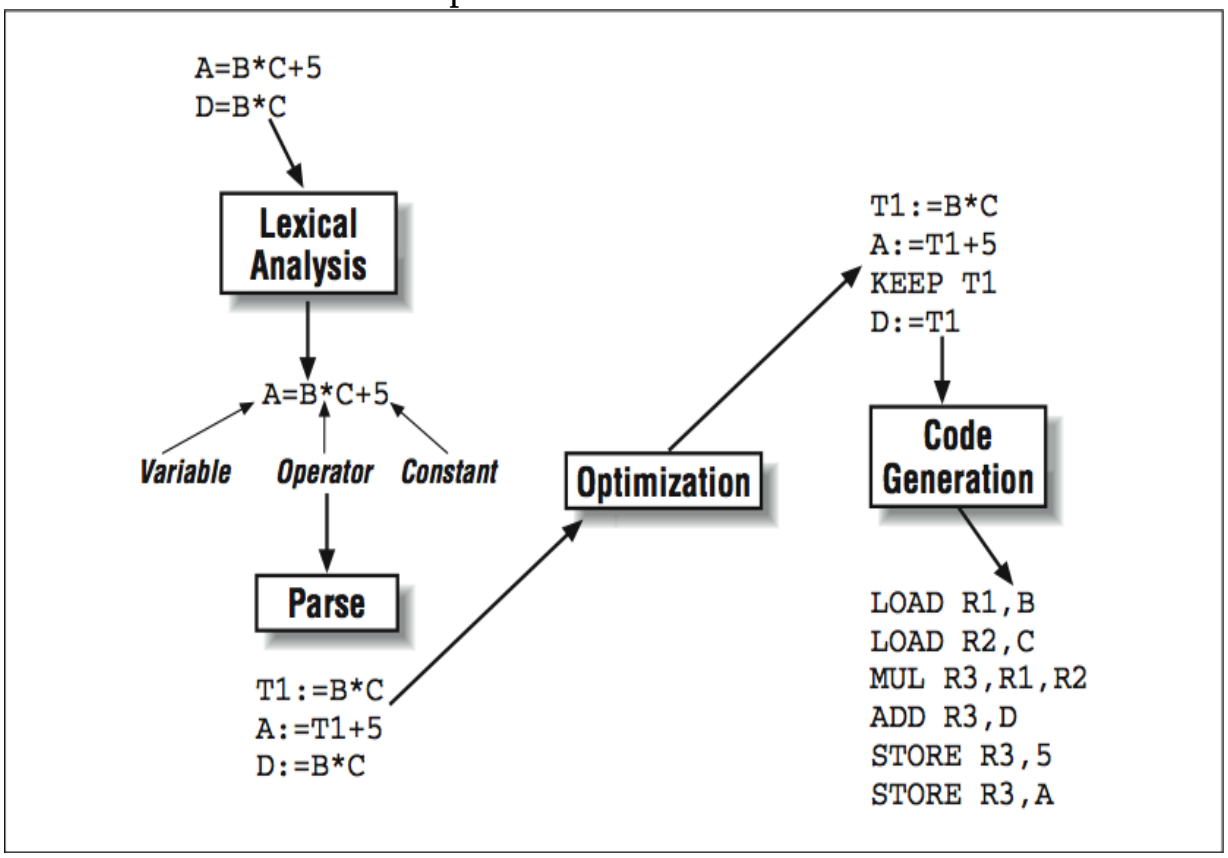
incrementales. En cierto modo el retador potencial más fuerte a FORTRAN a largo plazo puede venir en la forma de un conjunto de herramientas numéricas como Matlab. Sin embargo, los paquetes como Matlab tienen su propio conjunto de retos de optimización que deben superarse antes de que puedan derrocar a FORTRAN 77.

Qué Hace un Compilador - Visita Guiada al Optimizador de Código

Comenzaremos dando un paseo por el optimizador de código de un compilador, para verlo en acción. Creemos que es interesante, y si puede usted volverse empático con el compilador se convertirá en un mejor programador; verá lo que el compilador espera de usted, así como lo que puede hacer por sí mismo.

El Proceso de Compilación

Procesos básicos de un compilador



Típicamente, el proceso de compilación se divide en un número de pasos identificables, como se muestra en la [link]. Aunque no todos los compiladores estén implementados exactamente de esta manera, ayuda a comprender las diferentes funciones que éstos deben realizar:

1. Una fase de precompilación o preprocesamiento, donde se realiza cierta manipulación textual sencilla del código fuente. El paso de

preprocesamiento puede procesar o incluir archivos, así como realizar substituciones sencillas de cadenas de texto a lo largo del código.
2. La fase de análisis lexicográfico es la que descompone las sentencias de código fuente entrantes en tokens tales como variables, constantes, comentarios o elementos del lenguaje.
3. La fase de análisis sintáctico es la encargada de comprobar la sintaxis de la entrada, y el copilador traduce el programa entrante a un lenguaje intermedio que está listo para su optimización.
4. Se le realizan una o más pasadas de optimización al lenguaje intermedio.
5. Un generador de código objeto traduce el lenguaje intermedio a código ensamblador, tomando en consideración los detalles arquitectónicos particulares del procesador en cuestión.

Conforme los compiladores se vuelven más y más sofisticados para poder dar hasta el último bit de rendimiento del procesador, algunos de estos pasos (especialmente los de optimización y generación de código) se vuelven más y más confusos. En este capítulo nos enfocaremos en el optimizador de código tradicional de un compilador, y en los capítulos subsecuentes revisaremos más de cerca cómo hacen optimizaciones más sofisticadas los compiladores modernos.

Representación en Lenguaje Intermedio

Como estamos más interesados en la optimización de nuestro programa, comenzaremos nuestra discusión a partir de la salida de la fase de análisis sintáctico del compilador. Dicha salida está en la forma de un *lenguaje intermedio* (LI), algo entre un lenguaje de alto nivel y un lenguaje ensamblador. El lenguaje intermedio expresa los mismos cálculos que en el programa original, en una forma que el compilador pueda manipular más fácilmente. Es más, ciertas instrucciones que no están presentes en el fuente, tales como expresiones de direccionamiento para referencias a arreglos, se hacen visibles junto con el resto del programa, haciéndolo también de este modo sujeto de optimización.

¿Cómo luce un lenguaje intermedio? En términos de complejidad, es similar a un código ensamblador, pero no tan simple como para que se

pierdan las definiciones[footnote] y usos de las variables. Necesitaremos la información acerca de definición y uso para analizar el flujo de datos a través del programa. Típicamente los cálculos se expresan como u flujo de *cuádruplas* — sentencias con exactamente un operador, (hasta) dos operandos, y un resultado.[footnote] Presuponiendo que cualquier cosa en el programa fuente original pueda cambiar su representación en términos de cuádruplas, tenemos un lenguaje intermedio utilizable. Para darnos una idea de cómo trabaja, reescribiremos la sentencia siguiente como una serie de cuatro cuádruplas:

Por "definiciones" nos referimos a la asignación de valores, no a las declaraciones.

De manera más general, el código puede representarse como *n*-tuplas. Depende del nivel del lenguaje intermedio.

```
A = -B + C * D / E
```

Tomando todo como una unidad, la sentencia tiene cuatro operadores y cuatro operandos: `/`, `*`, `+` y `-` (negación), y `B`, `C`, `D`, y `E`. Claramente es demasiado para que quepa en una cuádrupla. Necesitamos una forma con exactamente un operador y, cuando mucho, dos operandos por sentencia. La versión que sigue lo lleva a cabo, empleando variables temporales para almacenar los resultados intermedios:

```
T1 = D / E
T2 = C * T1
T3 = -B
A  = T3 + T2
```

Por supuesto, un lenguaje intermedio utilizable requiere de algunas otras características, como apuntadores. Estamos por sugerir la creación de nuestro propio lenguaje intermedio para investigar cómo trabajan las optimizaciones. Para comenzar, necesitamos establecer unas pocas reglas:

- Las instrucciones están formadas por un código de operación, dos operandos y un resultado. Dependiendo de la instrucción, los operandos pueden quedar vacíos.
- Las asignaciones adoptan la forma `X := Y op Z`, que significa `X` optiene el resultado de `op` aplicado a `Y` y `Z`.
- Todas las referencias a memoria son cargas explícitas desde, o bien almacenamiento a, variables "temporales" `t`*n*.
- Los valores lógicos usados en las bifurcaciones se calculan separadamente del salto actual.
- Los saltos van a direcciones absolutas.

Si estamos construyendo un compilador, deberemos ser un poco más específicos. Para nuestros propósitos con esto basta. Considere el siguiente fragmento de código en C:

```
while (j < n) {
    k = k + j * 2;
    m = j * 2;
    j++;
}
```

Este ciclo se traduce en la representación en lenguaje intermedio que se muestra a continuación:

```
A:: t1  := j
    t2  := n
    t3  := t1 < t2
    jmp (B) t3
    jmp (C) TRUE

B:: t4  := k
    t5  := j
```

```
        t6  := t5 * 2
        t7  := t4 + t6
        k   := t7
        t8  := j
        t9  := t8 * 2
        m   := t9
        t10 := j
        t11 := t10 + 1
        j   := t11
        jmp (A) TRUE
    C::
```

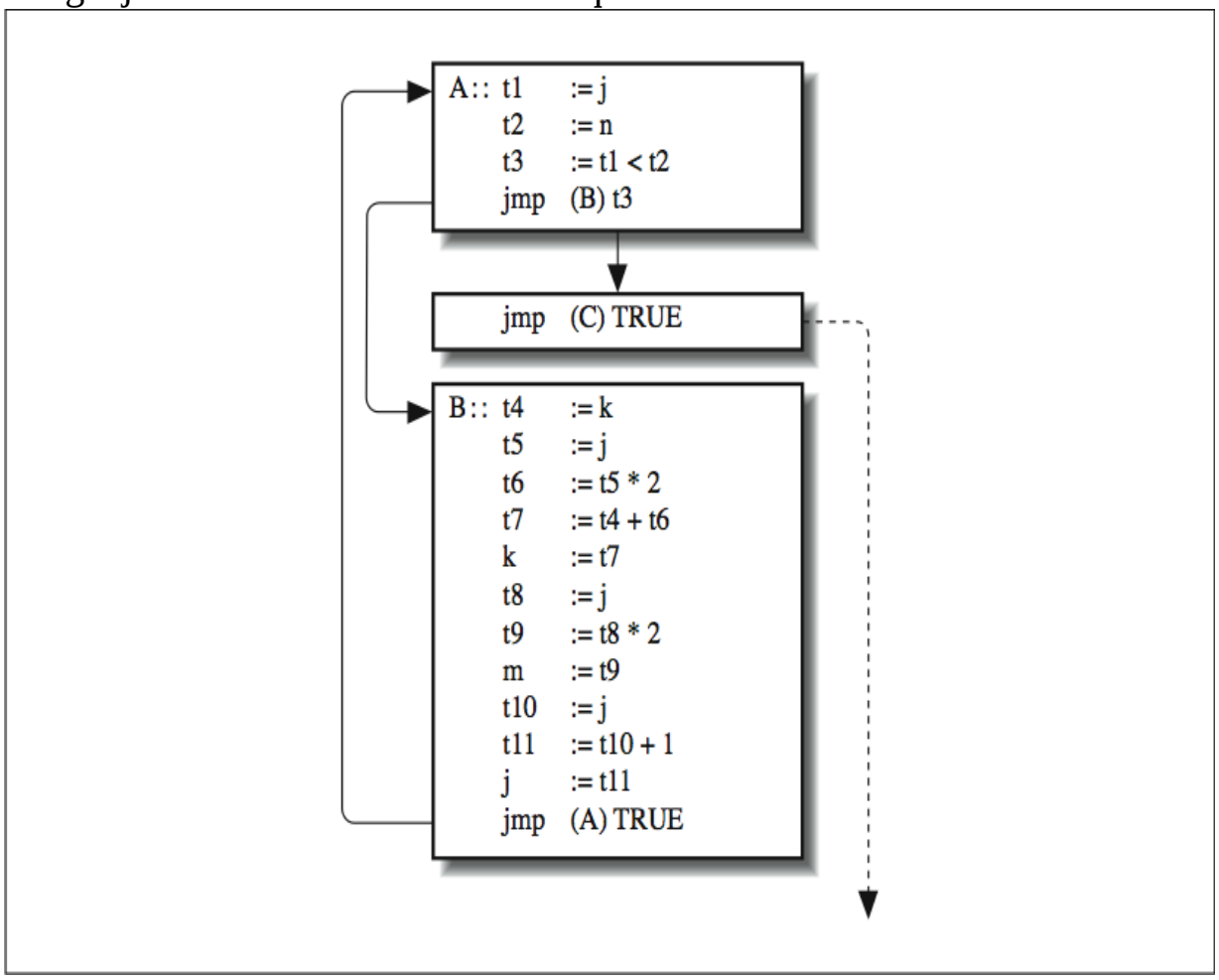
Cada línea de código fuente en C se representa mediante varias sentencias en LI. En muchos procesadores RISC, nuestro código LI es tan parecido al lenguaje máquina que podemos traducirlo directamente a código objeto. [footnote] A menudo el nivel más bajo de optimización consiste en una traducción literal del lenguaje intermedio a código máquina. Cuando esto sucede, el código generalmente es muy largo y su rendimiento es pobre. Al revisarlo, puede usted encontrar lugares donde ahorrar unas pocas instrucciones. Por ejemplo, `j` se carga en variables temporales en cuatro lugares; seguramente podemos reducirlo. Tenemos que realizar algo de análisis y algunas optimizaciones.
Véase [link] para algunos ejemplos de código máquina obtenido directamente a partir de lenguaje intermedio.

Bloques Básicos

Tras generar nuestro lenguaje intermedio, queremos cortarlo en *bloques básicos*. Se trata de secuencias de código que comienzan con una instrucción que o bien sigue una bifurcación o es en sí misma el destino de un salto. Dicho de otra forma, cada bloque básico tiene una entrada (en la parte superior) y una salida (en la parte inferior). [link] representa nuestro código LI como un grupo de tres bloques básicos. Estos bloques hacen el código más fácil de analizar. Al restringir el flujo de control al interior de un bloque básico de arriba hacia abajo y eliminar todas las bifurcaciones, podemos asegurarnos que si se ejecuta la primera sentencia, también lo hará

la segunda, y así sucesivamente. Por supuesto, las bifurcaciones no han desaparecido, pero las hemos forzado afuera de los bloques en la forma de flechas de conexión - el *grafo de flujo*.

Lenguaje intermedio dividido en bloques básicos



Ahora somos libres de extraer información de los bloques mismos. Por ejemplo, podemos decir con certeza cuáles variables usa cierto bloque, y cuáles define (les asigna valor), cosa que no hubiéramos podido ser capaces de hacer si el bloque contuviera una bifurcación. También podemos reunir la misma clase de información sobre los cálculos que realizan. Tras haber analizado los bloques, de forma tal que sepamos qué entra y qué sale de ellos, podemos modificarlos para mejorar el rendimiento, sin preocuparnos acerca de la interacción entre bloques.

Qué Hace un Compilador - Niveles de Optimización

Existe una amplia variedad de técnicas de optimización, no todas ellas aplicables en todas las circunstancias. Así que normalmente se le dan al usuario algunas opciones sobre cuáles optimizaciones realizar y cuáles no. A menudo esto se expresa en la forma de un *nivel de optimización* que se le especifica al compilador como una opción de la línea de comandos, como por ejemplo –O3.

Entre los distintos niveles de optimización controlados mediante una bandera suelen estar los siguientes:

- *Ninguna optimización*: Genera código máquina a partir directamente del código intermedio, que puede ser un código muy largo y lento. Se usa primordialmente para los depuradores, y para establecer la salida correcta del programa. Dado que cada operación se realiza precisamente como el usuario lo especificó, debe ser correcta.
- *Optimizaciones básicas*:Similares a las descritas en este capítulo. Generalmente trabaja minimizando el lenguaje intermedio y generando código compacto y rápido.
- *Análisis interprocedimental*: Observa más allá de las fronteras de una única rutina, en busca de oportunidades de optimización. Este nivel de optimización puede incluir extender una optimización bñásica, tal como la propagación de copias, a través de múltiples rutinas. Otro resultado de esta técnica es la inserción de procedimientos en línea (inline) en aquellos lugares donde mejore el rendimiento.
- *Análisis de perfil a tiempo de ejecución*: Es posible usar el perfilado a tiempo de ejecución para ayudar al compilador a generar código mejorado, basado en su conocimiento de los patrones de ejecución a tiempo de ejecución, reunidos a partir de la información del perfil.
- *Optimizaciones de punto flotante*: El estándar de punto flotante del IEEE (IEEE 754) especifica con precisión cómo se llevan a cabo las operaciones de punto flotante. El compilador puede identificar ciertas transformaciones algebraicas que incrementan la velocidad del programa (tales como reemplazar una división con un recíproco una multiplicación), pero ello pudiera cambiar los resultados de salida respecto al código no optimizado.

- *Análisis de flujo de datos*: Identifica el paralelismo potencial entre instrucciones, bloques o incluso iteraciones sucesivas de bucles.
- *Optimización avanzada*: Puede incluir vectorización, paralelización o descomposición de datos automáticas en computadoras con arquitecturas avanzadas.

Tales optimizaciones pueden controlarse mediante varias opciones distintas del compilador. A menudo se lleva algún tiempo cavilar la mejor combinación de banderas de compilación para un código o conjunto de códigos en particular. En algunos casos, los programadores compilan diferentes rutinas usando diferentes configuraciones de optimización, para lograr el mejor rendimiento global.

Qué Hace un Compilador - Optimizaciones Clásicas

Una vez dividido el lenguaje intermedio en bloques básicos, hay diversas optimizaciones que pueden realizarse sobre el código de tales bloques. Algunas son muy simples y afectan a unas pocas tuplas dentro de un bloque básico. Otras mueven código de un bloque básico a otro, sin alterar los resultados del programa. Por ejemplo, a menudo resulta útil mover una cálculo desde el cuerpo de un ciclo al código que precede de forma inmediata al bucle.

En esta sección, vamos a listar por nombre las optimizaciones clásicas, e indicarle a usted para qué son. No estamos sugiriendo que *usted* haga los cambios; la mayoría de lso compiladores desde mitad de la década de 1980 realizan de modo completamente automático tales optimizaciones, excepto en el nivel más bajo de optimización. Como vimos a inicios del capítulo, si comprende usted aquello que pueden (y que no pueden) hacer los compiladores, se convertirá en un mejor programador porque será capaz de jugar con las fortalezas de ellos.

Propagación de Copia

Para iniciar, veamos una técnica para desenredar cálculos. Échele un vistazo al siguiente segmento de código: observe los dos cálculos que involucran a `X`.

```
X = Y
Z = 1.0 + X
```

Tal como está escrito, la segunda sentencia requiere del resultado de la primera antes de que pueda ejecutarse -requiere usted de `X` para calcular `Z`. Las dependencias innecesarias a menudo se traducen en retrasos a tiempo de ejecución. [footnote] Realizando algunos reacomodos, podemos hacer que la segunda sentencia sea independiente de la primera, al *propagar* una copia de `Y`. El nuevo cálculo para `Z` usa el valor de `Y` directamente:

Este código es un ejemplo de dependencia de flujo. Describo las dependencias en detalle en [link].

```
X = Y
Z = 1.0 + Y
```

Observe que dejamos intacta la sentencia `X=Y`. Puede que usted se pregunte, "¿por qué dejarla así?" El problema es que no podemos decir en qué otras partes se requiere el valor de `X`. Eso es algo que deberá decidir otro análisis. Si resulta que ninguna otra sentencia requiere del valor de `X`, la asignación se eliminará posteriormente, por medio de la remoción de código muerto.

Plegado de Constantes

Un compilador inteligente puede encontrar constantes a lo largo de su programa. Algunas de ellas son “obvias”, como aquellas definidas en las sentencias de parámetros. Otras resultan menos obvias, tales como las variables locales que nunca se redefinen. Cuando las combina en un cálculo, obtiene una *expresión constante*. El pequeño programa siguiente tiene dos constantes, `I` y `K`:

```
PROGRAM MAIN
INTEGER I,K
PARAMETER (I = 100)
K = 200
J = I + K
END
```

Dado que tanto `I` como `K` son constantes individualmente, la combinación de `I+K` es constante, lo cuál significa que `J` también es una constante. El

compilador reduce las expresiones constantes como `I+K` a constantes, mediante una técnica llamada *plegado de constantes*.

¿Cómo funciona el plegado de constantes? Puede ver que es posible examinar cada camino en el cuál puede definirse una variable en ruta hacia un bloque básico particular. Si descubre usted que todos los caminos se inician en el mismo valor, éste es constante; puede reemplazar todas las referencias a esa variable con una constante. Este reemplazo tiene un efecto de rizado a lo largo de la ruta. Si el compilador se percata de que encontró una expresión compuesta solamente por constantes, puede evaluarla a tiempo de compilación y reemplazarla por una constante. Tras varias iteraciones, el compilador habrá localizado la mayoría de las expresiones que son candidatas al plegado de constantes.

A veces un programador puede mejorar el rendimiento, al hacer al compilador consciente de los valores constantes en su aplicación. Por ejemplo, en el siguiente segmento de código:

```
X = X * Y
```

el compilador generará código muy diferente a tiempo de ejecución si sabe que `Y` valía 0, 1, 2, or 175.32. Si no conoce el valor para `Y`, debe generar la secuencia de código más conservadora (no necesariamente la más rápida). Un programador puede comunicarle esos valores a través del uso de la sentencia `PARAMETER` en FORTRAN. Mediante el empleo de dicha sentencia, el compilador conoce los valores para estas constantes a tiempo de ejecución. Otro ejemplo que hemos visto es:

```
DO I = 1,10000
  DO J=1,IDIM
    .....
  ENDDO
ENDDO
```

Tras revisar el código, es claro que `IDIM` era uno de los valores 1, 2, or 3, dependiendo del conjunto de datos en uso. Claramente si el compilador sabía que `IDIM` valía 1, puede generar un código mucho más simple y rápido.

Remoción de Código Muerto

A menudo los programas contienen secciones de *código muerto* que no tiene efecto en las respuestas, y que por tanto puede quitarse. Ocasionalmente, quien escribe el código muerto en el programa es el autor, pero es más común que sea el propio compilador quien lo hace; muchas optimizaciones producen código muerto que hay que barrer después.

Hay dos tipos de código muerto:

- Instrucciones inalcanzables
- Instrucciones que producen resultados que jamás se usan

Es fácil que usted escriba código inalcanzable en su programa, al hacer que el flujo de control pase alrededor de él - permanentemente. Si el compilador puede decir que es inalcanzable, lo eliminará. Por ejemplo, es imposible alcanzar la sentencia `I = 4` en este programa:

```
PROGRAM MAIN
I = 2
WRITE (*,*) I
STOP
I = 4
WRITE (*,*) I
END
```

El compilador desechará todo hasta la sentencia `STOP`, y probablemente le envíe un aviso. El código inalcanzable producido por el compilador durante la optimización se retirará silenciosamente.

Los cálculos con variables locales pueden producir resultados que nunca se usan. Al analizar la definición y usos de una variable, el compilador puede ver qué otras partes de la rutina la referencian. Por supuesto el compilador no puede decir el destino último de las variables que se pasan entre rutinas, externas o comunes, así que tales cálculos siempre se mantienen (siempre y cuando sean alcanzables).[footnote] En el siguiente programa, los cálculos que involucran a `k` no contribuyen en absoluto a la respuesta final, y son buenos candidatos para la eliminación de código muerto:
Si un compilador realiza un análisis interprocedimental suficiente, puede incluso optimizar variables entre fronteras de rutinas. El análisis interprocedimental puede ser la perdición para los códigos de bancos de pruebas si trata de medir el tiempo de un cálculo sin usar los resultados del mismo.

```
main ()
{
  int i,k;
  i = k = 1;
  i += 1;
  k += 2;
  printf ("%d\n",i);
}
```

La eliminación de código muerto a menudo produce algunos resultados sorprendentes con bancos de pruebas pobremente escritos. Véase [link] para un ejemplo de este tipo de código.

Reducción de Intensidad

Las operaciones o expresiones tienen costos de tiempo asociados. A veces es posible reemplazar un cálculo muy caro por otro más económico. Llamamos a esto *reducción de intensidad*. El siguiente fragmento de código contiene dos operaciones caras:

```
REAL  X,Y
Y  =  X**2
J  =  K*2
```

En operaciones de exponenciación como la de la primera línea, el compilador generalmente incrusta una llamada a una rutina matemática de biblioteca. En dicha rutina, `X` se convierte a un logaritmo, se multiplica y luego se reconvierte. Sobre todo, elevar `X` a una potencia resulta caro - quizás tome cientos de ciclos de máquina. La clave es observar que `X` se está elevando a una potencia entera y pequeña. Una alternativa mucho más económica consiste en expresarlo como `X*X`, y pagar sólo el costo de una multiplicación. La segunda sentencia muestra la multiplicación entera de una variable `K` por 2. Sumar `K+K` da el mismo resultado, pero toma menos tiempo.

Hay muchas oportunidades para reducción de intensidad generada por el compilador; éstas son sólo dos de ellas. Veremos un caso especialmente importante cuando analicemos la simplificación de variable por inducción. Otro ejemplo de una reducción de intensidad es reemplazar las multiplicaciones de enteros potencia de dos por desplazamientos lógicos.

Renombramiento de Variables

En [link], hablamos acerca de renombrar registros. Algunos procesadores pueden tomar decisiones a tiempo de ejecución para reemplazar todas las referencias al registro 1 por referencias al registro 2, por ejemplo, para evitar cuellos de botella. El renombramiento de registros evita que las instrucciones que están reciclando los mismos registros para propósitos diferentes, tengan que esperar hasta que las instrucciones previas hayan dejado de usarlos.

La misma situación puede ocurrir en los programas - la misma variable (i.e. posición de memoria) puede reciclarse para dos propósitos no relacionados. Por ejemplo, observe la variable `X` en el siguiente fragmento:

```
x = y * z;
q = r + x + x;
x = a + b;
```

Cuando el compilador reconoce que se está reciclando una variable, esto es que sus usos actual y futuros son independientes, puede sustituirla por una nueva variable para mantener los cálculos separados:

```
x0 = y * z;
q = r + x0 + x0;
x = a + b;
```

El *renombramiento de variables* es una técnica importante, porque deja en claro cuáles cálculos son independientes los unos de los otros, lo cuál mejora el número de cosas que pueden hacerse en paralelo.

Eliminación de Subexpresiones Comunes

Las subexpresiones son partes de expresiones más grandes. Por ejemplo, `A+B` es una subexpresión de `C*(A+B)`. Si `A+B` aparece en varios lugares, como sucede a continuación, le llamamos una *subexpresión común*:

```
D = C * (A + B)
E = (A + B)/2.
```

En vez de calcular `A + B` dos veces, el compilador puede generar una variable temporal y usarla dondequiera que se necesite `A + B`:

```
temp = A + B
D = C * temp
E = temp/2.
```

Algunos compiladores se esfuerzan más que otros en encontrar subexpresiones comunes. Se reconocen la mayoría de los pares, tales como `A+B`. Algunos pueden reconocer la reutilización de intrínsecos, tales como `SIN(X)`. Pero no espere que el compilador vaya mucho más allá. Subexpresiones como `A+B+C` no son computacionalmente equivalentes a formas reasociadas como `B+C+A`, aunque sean algebraicamente equivalentes. Con el fin de proporcionar resultados predecibles en los cálculos, FORTRAN debe o bien realizar las operaciones en el orden especificado por el usuario, o bien reordenarlos en una forma que garantice exactamente el mismo resultado. A veces el usuario no se preocupa de la forma en que se asocia `A+B+C`, pero el compilador no puede asumir que al usuario no le preocupa.

El cálculo de direcciones proporciona una oportunidad particularmente rica para la eliminación de subexpresiones comunes. Usted no requiere ver los cálculos en el código fuente, pues los genera el compilador. Por ejemplo, una referencia al elemento de un arreglo `A(I,J)` puede traducirse a una expresión en lenguaje intermedio tal como:

```
address(A) + (I-1)*sizeof_datatype(A)
+ (J-1)*sizeof_datatype(A) *
column_dimension(A)
```

Si `A(I,J)` se usa más de una vez, tenemos múltiples copias del mismo cálculo de direcciones. La eliminación de subexpresiones comunes descubrirá y agrupará (con un poco de suerte) los cálculos redundantes.

Remoción de Código Invariante en Bucles

Es en los bucles donde los programas de cómputo de alto rendimiento gastan la mayoría de su tiempo. El compilador busca cada oportunidad que tiene de mover cálculos fuera del cuerpo de un bucle hacia el código que lo rodea. Las expresiones que no cambian después de haber ingresado al ciclo (*expresiones invariantes en el bucle*) son la primera opción. El siguiente ciclo tiene dos expresiones invariantes:

```
DO I=1,N
  A(I) = B(I) + C * D
  E = G(K)
ENDDO
```

A continuación, hemos modificado las expresiones para mostrar cómo pueden moverse afuera del ciclo:

```
temp = C * D
DO I=1,N
  A(I) = B(I) + temp
ENDDO
E = G(K)
```

Es posible mover el código antes o después del cuerpo del bucle. Como sucede con la eliminación de subexpresiones comunes, la aritmética de direcciones resulta un blanco particularmente importante para la técnica de movimiento de invariantes en los bucles. Las porciones que cambian lentamente en el cálculo de los índices pueden enviarse a los suburbios, para ejecutarse sólo cuando se requieran.

Simplificación de Variables de Inducción

Los ciclos pueden contener lo que se conoce como *variables de inducción*. Su valor cambia como una función lineal del contador de iteraciones del ciclo. Por ejemplo, `K` es una variable de inducción en el siguiente bucle. Su valor está ligado al índice del ciclo:

```
DO I=1,N
  K = I*4 + M
   ...
ENDDO
```

La *simplificación de variables de inducción* reemplaza los cálculos que involucran variables como `K` por otras más simples. Dado un punto de inicio y la primera derivada de la expresión, puede usted llegar al valor de `K` para la *n*-ésima iteración, recorriendo paso a paso las *n-1* iteraciones que intervienen:

```
K = M
DO I=1,N
  K = K + 4
   ...
ENDDO
```

Ambas formas del bucle no son equivalentes; la segunda no le proporciona a usted el valor de `K`, dado cualquier valor de `I`. Dado que no podemos saltar en la mitad del bucle a la *n*-ésima iteración, `K` siempre toma los mismos valores que hubiera tenido si hubiéramos dejado la expresión original.

Las simplificación de variables de inducción probablemente no sea una optimización muy importante, excepto que el cálculo de direcciones de los arreglos se parece mucho al cálculo de `K` en el ejemplo anterior. Por

ejemplo, el cálculo de direcciones para `A(I)` dentro de un bucle iterando sobre la variable `I` se ve más o menos así:

```
address = base_address(A) + (I-1) *
sizeof_datatype(A)
```

Realizar toda esta matemática es innecesario. El compilador puede crear una nueva variable de inducción para las referencias a A y simplificar el cálculo de direcciones:

```
      afuera del ciclo...
      address = base_address(A) - (1 *
sizeof_datatype(A))
      dentro del ciclo...
      address = address + sizeof_datatype(A)
```

La simplificación de variables de inducción resulta especialmente útil en aquellos procesadores que pueden incrementar automáticamente un registro cada vez que se utiliza un apuntador para una referencia a memoria. Mientras se recorre el ciclo paso a paso, tanto la referencia a memoria como la aritmética de direcciones pueden exprimirse en una sola instrucción -un gran ahorro.

Generación de Código Objeto

La precompilación, los análisis lexicográfico y sintáctico y muchas técnicas de optimización son a menudo transportables, pero la generación de código es muy específica del procesador destino. En cierta forma es en esta fase donde los compiladores desquitan su precio, en los sistemas RISC de un solo procesador.

Cualquier cosa que no se haga en hardware debe realizarse en software. Ello significa que si el procesador no puede resolver conflictos de recursos,

tales como la sobreutilización de un registro o fila de procesamiento, el compilador deberá hacerse cargo del asunto. Permitir que el compilador se ocupe de ello no es necesariamente malo, más bien es una decisión de diseño. Un compilador complicado y un hardware simple y rápido pueden tener una alta relación costo/beneficio para ciertas aplicaciones. Dos procesadores que se encuentran en los extremos de este espectro son el MIPS R2000 y el HP PA-8000. El primero depende fuertemente del compilador para planificar las instrucciones y distribuir los recursos equitativamente. El segundo administra ambas cosas a tiempo de ejecución, aunque ambos dependan del compilador para proporcionar una mezcla de instrucciones equitativa.

En todas las computadoras, la selección de registros es un reto porque, dependiendo de su número, los registros son recursos preciosos. Usted quiere asegurarse que las variables más activas residan permanentemente en los registros, a expensas de otras. En aquellas máquinas sin renombramiento de registros (véase [link]), usted debe asegurarse que el compilador no trate de reciclar los registros demasiado rápido, o de otra forma el procesador tendrá que retrasar los cálculos, en espera de que alguno de ellos se libere.

Algunas instrucciones del repertorio también hacen que su compilador no tenga que ejecutar otras. Ejemplo de esto son el autoincremento de los registros que se usan como índices de arreglos, o el uso de asignaciones condicionales en vez de bifurcaciones. Ambas ahorran al procesador realizar cálculos extras, produciendo un flujo de instrucciones más compacto.

Finalmente, están las oportunidades para incrementar el paralelismo. Los programadores generalmente piensan serialmente, especificando pasos en sucesión lógica. Desafortunadamente, el código fuente serial produce código objeto serial. Un compilador que aspire a usar eficientemente el paralelismo del procesador, deberá ser capaz de mover instrucciones y encontrar operaciones que puedan realizarse simultáneamente. Este es uno de los mayores retos de los creadores de compiladores actualmente. Conforme los diseños superescalares y de *tamaño de instrucción muy grande* (VLIW, Very Long Instruction Word) se vuelven capaces de ejecutar

más instrucciones por ciclo de reloj, el compilador tendrá que excavar más profundamente en busca de operaciones que puedan ejecutarse a la vez.

Qué Hace un Compilador - Notas de Cierre

Este capítulo ha sido una introducción básica a la forma en que funciona el optimizador de código de un compilador. Sin embargo, no es lo último que diremos acerca de los compiladores. Con el objeto de realizar la vectorización, paralelización y descomposición de datos automáticamente, los compiladores adicionalmente deben analizar el código fuente. Conforme nos topemos con estos tópicos, iremos discutiendo el impacto sobre y del compilador, y cómo los programadores pueden interactuar mejor con éste.

En las arquitecturas RISC modernas de un solo procesador, los compiladores por lo general producen mejor código que la mayoría de los programadores humanos en ensamblador. En vez de tratar de compensar el trabajo de un compilador simplista agregando optimizaciones a mano, nosotros como programadores debemos mantener nuestros programas simples, de forma que no confundan a éste. Al entender los patrones que los compiladores son capaces de optimizar, podemos enfocarnos en escribir programas sencillos que sean transportables y comprensibles.

Qué Hace un Compilador - Ejercicios

Exercise:

Problem:

¿Su compilador reconoce el código muerto en el siguiente programa? ¿Cómo puede estar seguro? ¿Le envía el compilador algún aviso?

```
main()
{
    int k=1;
    if (k == 0)
        printf ("Esta sentencia nunca se ejecuta.\n");
}
```

Exercise:

Problem:

Compile el siguiente código, y ejecútelo bajo distintos niveles de optimización.

Trate de adivinar los distintos tipos de optimizaciones que se están realizando para mejorar el rendimiento, conforme se incrementa el nivel de optimización.

```
REAL*8 A(1000000)
DO I=1,1000000
   A(I) = 3.1415927
ENDDO
DO I=1,1000000
  A(I) = A(I) * SIN(A(I)) + COS(A(I))
ENDDO
```

```
PRINT *,"Terminado"
```

Exercise:

Problem:

Tome el siguiente segmento de código y compílelo a varios niveles de optimización. Observe el código ensamblador generado (en algunos compiladores esto se hace con la opción -S) y encuentre los efectos de cada nivel de optimización sobre el lenguaje máquina. Mida el tiempo de ejecución del programa para ver el rendimiento a diferentes niveles de optimización. Si tiene acceso a múltiples arquitecturas, observe el código generado usando los mismos niveles de optimización sobre diferentes arquitecturas.

```
REAL*8 A(1000000)
COMMON/BLK/A
 .... Llamada para medir el tiempo
DO I=1,1000000
  A(I) = A(I) + 1.234
ENDDO
 .... Llamada para medir el tiempo
END
```

¿Por qué es necesario poner el arreglo adentro de un bloque común?

Cronometraje y Perfilado - Introducción

Tal vez con hacer que su código produzca las respuestas correctas sea suficiente. Después de todo, si sólo planea usar el programa de vez en cuando, o si sólo toma unos minutos en ejecutarse, el tiempo de ejecución no es un asunto que le importe demasiado. Pero puede que no siempre ese sea el caso. Típicamente, la gente se preocupa por el tiempo de ejecución de sus programas por una de dos razones:

- La carga de trabajo ha crecido.
- Están considerando una nueva máquina.

Resulta claro por qué debe usted preocuparse acerca del rendimiento de su programa si la carga de trabajo crece. Tratar de atiborrar 25 horas de tiempo de cómputo en un día de 24 horas es una pesadilla administrativa. Pero, ¿por qué debiera preocuparse la gente que está considerando una nueva máquina, acerca del tiempo de ejecución? Después de todo, la nueva máquina presumiblemente será más rápida que la antigua, así que todo debiera tomar menos tiempo. La razón es que cuando la gente está evaluando nuevas máquinas, requieren una base de comparación -un banco de pruebas. La gente a menudo usa programas que le resultan familiares como banco de pruebas. Tiene sentido: usted quiere un banco de prueba que sea representativo de la clase de trabajos que realiza, y nada es más representativo del trabajo que usted hace... ¡que el trabajo que usted hace!

Llevar a cabo bancos de pruebas suena fácil, suponiendo que tenga usted herramientas de cronometraje. [footnote] Usted sólo quiere asegurarse que lo que tales herramientas estén reportando sea lo mismo que usted piensa que está obteniendo; especialmente si nunca las ha utilizado con anterioridad. Para ilustrar el punto, imagine si alguien tomase su reloj y lo reemplazara con otro que expresa el tiempo en un sistema de unidades exótico, o con tres conjuntos de manecillas sobrepuestas. Resultaría muy confuso; tendría problemas leyéndolo. Estaría justificablemente nervioso de tener que conducir sus asuntos mediante un reloj que no comprende.
El tiempo es dinero.

Las herramientas de cronometraje de UNIX son como el reloj de seis agujas, reportando tres tipos diferentes de medidas. No proporcionan

información incongruente, sólo presentan más información de la que puede usted expresar mediante un único valor numérico. Nuevamente, el truco consiste en aprender a leer este reloj. Y eso es lo que hace la primera parte de este capítulo. Investigaremos los diferentes tipos de medidas que determinan el desempeño de un programa.

Si está usted planeando afinar un programa, requiere más que sólo información de cronometraje. ¿En dónde se está gastando el tiempo, en un solo bucle, en una sobrecarga de llamado a subrutinas, o con problemas de memoria? Para los afinadores, las últimas secciones de este capítulo discutirán cómo perfilar el código a nivel procedimental y de sentencia. También discutiremos qué significan los perfiladores, y cómo predicen el enfoque que debe usted tomar si decide meterle mano al código en busca de mayor rendimiento, y qué posibilidades tendrá de obtener éxito.

Cronometraje y Perfilado - Cronometraje

Vamos a asumir que su programa se ejecuta correctamente. Resulta algo ridículo cronometrar un programa que no se está ejecutando bien, lo cual no quiere decir que no suceda. Dependiendo de lo que esté usted haciendo, puede estar interesado en saber cuánto tiempo gasta globalmente, o interesado sólo en una porción del programa. Le mostraremos cómo cronometrar primero el programa completo, y luego hablaremos acerca de cronometrar bucles o subrutinas individuales.

Cronometrando un Programa Completo

En UNIX, puede usted cronometrar la ejecución de un programa poniendo el comando *time* antes que cualquier otro que normalmente teclee en la línea de comandos. Cuando el programa termina, se produce un sumario de cronometraje. Por ejemplo, si su programa se llama *foo*, puede cronometrar su ejecución tecleando `time foo`. Si está usted usando el shell C o el shell Korn, *time* es uno de los comandos internos del shell. Con el shell Bourne, *time* es un comando separado, ejecutado desde */bin*. En cualquier caso, aparece la siguiente información al final de la ejecución:

- Tiempo en modo usuario
- Tiempo en modo sistema
- Tiempo transcurrido

Estas figuras de cronometraje resultan más fáciles de entender con algo de conocimiento previo. Conforme su programa se ejecuta, conmuta de ida y vuelta entre dos modos fundamentalmente diferentes: el *modo de usuario* y el *modo de kernel*. El estado de operación normal es el modo de usuario, en el cual se ejecutan las instrucciones que el compilador generó por usted, además de cualquier llamada a una subrutina de biblioteca enlazada con su programa. [footnote] Pudiera ser suficiente con la ejecución en modo de usuario siempre, excepto que los programas generalmente requieren otros servicios, tales como entrada/salida (I/O), y ello requiere la intervención del sistema operativo -el núcleo o kernel. Una solicitud de servicio del kernel hecha por su programa, o tal vez un evento externo al programa, causa la conmutación del modo de usuario al modo de kernel.

El tiempo de falla de caché también se consume aquí.

El tiempo empleado en la ejecución de cada uno de los dos modos se contabiliza por separado. La métrica del *tiempo de usuario* describe el tiempo gastado en el modo de usuario. Similarmente, la métrica de *tiempo de sistema* indica el tiempo gastado en modo de kernel. Conforme avanza el tiempo de usuario, cada programa en la máquina se contabiliza por separado. Esto es, no le contabilizarán a usted la actividad realizada por las aplicaciones de alguien más. El registro del tiempo de sistema funciona casi de la misma forma; sin embargo, bajo ciertas circunstancias, puede ser que le contabilicen a usted algunos servicios de sistema realizados a nombre de otras personas, además del suyo propio. Esta contabilidad incorrecta ocurre porque el programa de usted puede estar en ejecución al momento que alguna actividad externa causa una interrupción. No parece justo, pero consuélese con el hecho de que esto funciona así en ambos sentidos: puede que a los otros usuarios también les contabilicen la actividad de sistema de usted, por la misma razón.

Juntos, los tiempos de usuario y de sistema se conocen como *tiempo de CPU*. Generalmente, el tiempo de usuario es por mucho mayor que el tiempo de sistema. Esto es algo que cabe esperar, porque la mayoría de las aplicaciones sólo piden los servicios del sistema ocasionalmente. De hecho, un tiempo de sistema desproporcionadamente grande probablemente indique algún problema. Por ejemplo, aquellos programas que generan condiciones de excepción repetidamente, tales como fallos de página, referencias a memoria desalineadas, o excepciones de punto flotante, usan una cantidad de tiempo de sistema poco usual. El tiempo gastado en hacer cosas tales como buscar en disco, rebobinar una cinta, o esperar por caracteres provenientes de la terminal no se contabilizan en el tiempo de CPU. Ello se debe a que tales actividades no requieren de la CPU, que está disponible para suspenderlo y ejecutar otros programas.

La tercera pieza de información (correspondiente al tercer conjunto de manecillas del reloj) el *tiempo transcurrido*, es una medida del tiempo actual (tiempo de reloj de pared) que ha pasado desde que el programa inició. Para aquellos programas que gastan la mayoría de su tiempo calculando, el tiempo transcurrido debe ser muy parecido al tiempo de

CPU. Las razones por las cuales el tiempo transcurrido puede ser mayor son:

- Está usted compartiendo el tiempo de máquina con otros programas. [footnote]
  El comando *uptime* le proporciona una indicación aproximada del resto de la actividad en su máquina. Los últimos tres campos le indican el número promedio de procesos listos para ejecutarse durante los últimos 1, 5 y 15 minutos respectivamente
- Su aplicación lleva a cabo mucha I/O
- Su aplicación requiere más ancho de banda de memoria que la que está disponible en la máquina.
- Su programa está llevando a cabo paginación o intercambio.

La gente a menudo registra el tiempo de CPU y lo usa como un estimado del tiempo transcurrido. Usar el tiempo de CPU está bien cuando se trata de máquinas de una sola CPU, suponiendo que ha visto usted ejecutarse el programa cuando la máquina estaba tranquila y observó que ambos números eran muy parecidos. Pero para los multiprocesadores, el tiempo total de CPU puede ser muy diferente del tiempo transcurrido. Cada vez que haya dudas, espere hasta que tenga la máquina sólo para usted, y entonces cronometre su programa, usando el tiempo transcurrido. Es muy importante producir resultados de cronometría que puedan verificarse usando otra corrida, cuando se estén usando los resultados para tomar decisiones de compra importantes.

Si está ejecutando un UNIX derivado de Berkeley, el comando de cronómetro interno del shell C puede reportar otras estadísticas útiles. La forma por defecto de la salida que arroja se muestra en [link]. Revise la página del manual de su *csh* para ver más posibilidades.

Además de los datos de tiempos de CPU y utilizado, el comando *time* de *csh* produce información acerca del uso de la CPU, fallos de página, intercambios, operaciones de E/S bloqueadas (usualmente actividad de disco) y algunas medidas sobre cuánta memoria ocupaba nuestro programa cuando estaba en ejecución. Las describiremos una a la vez.

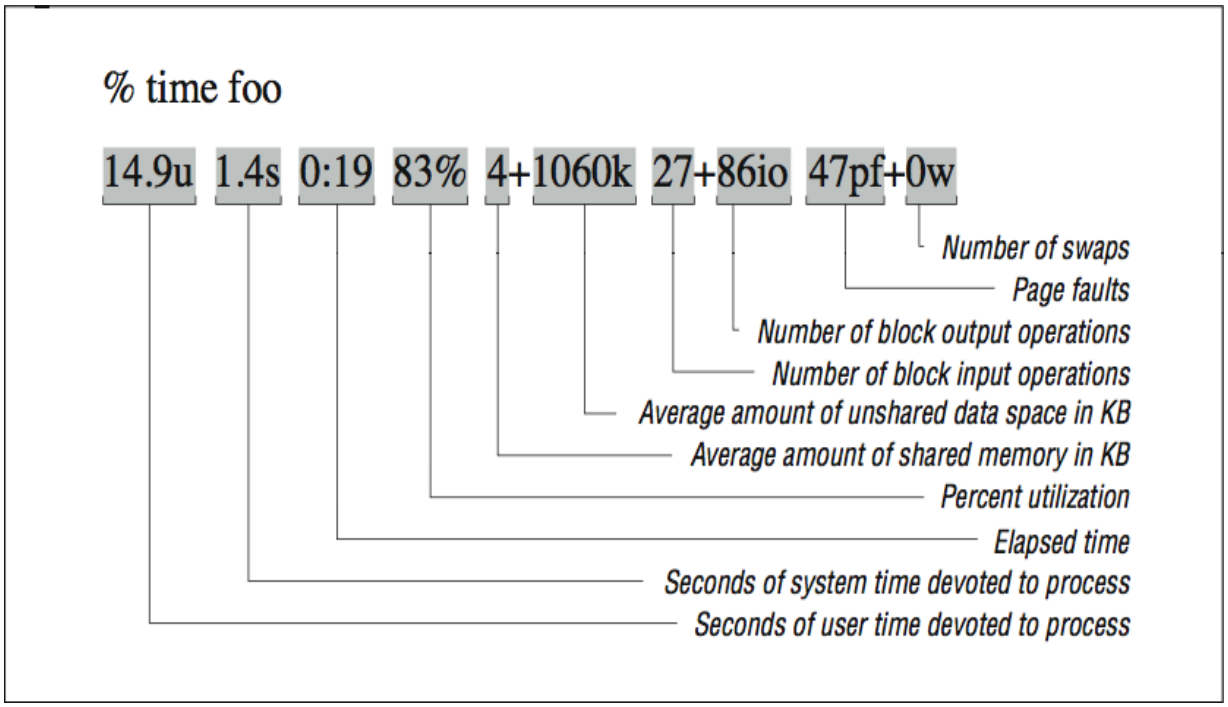
Porcentaje de Uso

El *porcentaje de uso* corresponde a la tasa de tiempo usado respecto al tiempo de CPU. Como mencionamos con anterioridad, existen varias razones por las que el uso de la CPU no sea del 100% o siquiera cercano. A menudo puede usted obtener un indicio, a partir de los otros campos, de cuál es el problema con su programa o si éste estaba compartiendo la máquina cuando lo ejecutaba.

Uso Promedio de Memoria Real

Las dos medidas de *uso promedio de la memoria* mostradas en [link] caracterizan los requerimientos de recursos del programa conforme se ejecuta.

La primera medida, el *espacio de memoria compartida*, contabiliza la cantidad promedio de memoria real que ocupa el segmento de texto de su programa - la porción que almacena las instrucciones de máquina. Se le lama "compartida" porque pueden compartirlo varias copias del programa que estén actualmente en ejecución (para ahorrar memoria). Años atrás, era posible que el segmento de texto consumiera una parte significativa del sistema de memoria, pero en estos días, con tamaños de memoria a partir de los 32 MB, tiene usted que compilar un programa fuente realmente enorme y usar cada bit de él para que la cantidad que figure en el uso de memoria compartida sea suficientemente grande como para causar preocupación. El requerimiento de espacio de memoria compartida es usualmente muy bajo, en comparación a la cantidad de memoria disponible en su máquina.
La función interna time de csh



La segunda medida promedio de uso de memoria, el *espacio de memoria no compartida*, describe el almacenamiento *real* dedicado a las estructuras de datos de su programa conforme se ejecuta. Este almacenamiento incluye las variables locales guardadas y las declaradas `COMMON` para `FORTRAN`, y las estáticas y externas de C. Resaltamos la palabra "real" aquí y arriba porque estos números nos hablan acerca del uso de memoria física, tomados a lo largo del tiempo. Puede ser que usted haya apartado arreglos con 1 trillón de elementos (espacio virtual), pero si su programa sólo se mueve sobre una esquina de tal espacio, sus requerimientos de memoria a tiempo de ejecución serán muy bajos.

Lo que no le dicen las mediciones de espacio de memoria no compartido, desafortunadamente, es la demanda pico de memoria de su programa. Una aplicación que requiere 100 MB durante 1/10 del tiempo y 1 KB el resto del tiempo parece requerir sólo 10 MB en promedio - lo cuál no es una imagen muy reveladora de los requerimientos de memoria del programa.

Operaciones de E/S Bloqueadas

Los dos datos para operaciones de *E/S bloqueadas* describen primordialmente el uso de disco, aunque los dispositivos de cintas y algunos otros periféricos pueden también usarse con E/S bloqueada. Las operaciones de E/S de caracteres, como la entrada y salida de la terminal, no aparecen aquí. Un gran número de operaciones de E/S bloqueadas puede explicar un uso de CPU menor del esperado.

Fallos de Página e Intercambios

Un número inusualmente alto de *fallos de página* o cualquier intercambio probablemente indica un sistema necesitado de memoria, lo cuál también puede explicar un tiempo transcurrido más largo de lo esperado. Puede ser que otros programas estén compitiendo por el mismo espacio. Y no olvide que incluso bajo condiciones óptimas, cada programa sufre de cierto número de fallos de página, como se explicó en [link]. En [link] se describen algunas técnicas para minimizar el número de fallos de página.

Cronometrando una Porción del Programa

Para algunos bancos de pruebas o esfuerzos de afinación, las medidas tomadas desde "afuera" del programa le indican a usted todo lo que necesita saber. Pero si está tratando de aislar las cifras de rendimiento de bucles o porciones de código individuales, puede que quiera incluir rutinas de cronometraje también al interior. La técnica básica es suficientemente sencilla:

1. Registrar el tiempo antes de comenzar a hacer X.
2. Hacer X.
3. Registrar el tiempo al completar X.
4. Restar el tiempo inicial del tiempo final.

Si, por ejemplo, el trabajo primario de X es calcular la posición de unas partículas, divida el tiempo total para obtener un número de posiciones de partículas por segundo. Pero debe ser cuidadoso: si realiza demasiados llamados a las rutinas de cronometraje, el observador se vuelve parte del experimento. Las rutinas de cronometraje también consumen tiempo, y su

sola presencia puede incrementar el número de fallos de cache de instrucciones o la paginación. Por otra parte, usted quiere que X tome una cantidad suficiente de tiempo en ejecutarse, como para que las mediciones sean útiles. Es muy importante poner atención al tiempo entre llamadas al cronómetro, porque el reloj usado por las funciones cronométricas tiene una resolución limitada. Un evento que ocurra dentro de una fracción de segundo es difícil de medir con precisión.

Obteniendo Información de Tiempos

En esta sección, discutiremos métodos para obtener varios valores de cronometraje durante la ejecución de su programa.

Para los programas FORTRAN, una biblioteca de funciones de cronometraje disponible en muchas máquinas se llama *etime*, que toma como argumento un arreglo de dos elementos `REAL*4`, y llena las celdas con el tiempo de CPU de usuario y el tiempo de CPU de sistema, respectivamente. El valor retornado por la función es la suma de los dos. He aquí un ejemplo de cómo se usa *etime* habitualmente:

```
      real*4 tarray(2), etime
      real*4 start, finish

      start = etime(tarray)
      finish = etime(tarray)

      write (*,*) 'Tiempo de CPU: ', finish -
start
```

No todos los vendedores proporcionan una función *etime*; de hecho, algunos no proporcionan rutinas de cronometraje para FORTRAN en absoluto. Pruébelo primero. Si obtiene un mensaje de símbolo indefinido cuando se enlace el programa, puede usar la siguiente rutina en C, que le proporciona la misma funcionalidad que *etime*:

```
      #include <sys/times.h>
      #define TICKS 100.

      float etime (parts)
      struct {
              float user;
              float system;
      } *parts;
      {
              struct tms local;
              times (&local);
              parts->user= (float)
local.tms_utime/TICKS;
              parts->system = (float)
local.tms_stime/TICKS;
              return (parts->user + parts-
>system);
      }
```

Hay un par de cosas que debe usted ajustar para hacerlo funcionar. Lo primero, para poder enlazar rutinas en C con rutinas en FORTRAN en su computadora, puede ser que deba agregar un guión bajo `(_)` tras el nombre de la función. Esto cambia la entrada a `float etime_ (parts)`. Además, puede que deba ajustar el parámetro `TICKS`. Asumimos que el reloj del sistema tenía una resolución de 1/100 de segundo (cierto en las máquinas Hewlett-Packard para las que se escribió esta versión de *etime*). 1/60 es muy común. En una RS-6000 el número debiera ser 1000. Puede encontrar el valor en un archivo llamado */usr/include/sys/param.h* en su máquina, o bien determinarlo empíricamente.

Abajo se muestra una rutina en C para recuperar la hora real, llamada *gettimeofday*. Está disponible para su uso ya sea en programas C o FORTRAN, y usa paso de parámetros por valor:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

void hpcwall(double *retval)
{
  static long zsec = 0;
  static long zusec = 0;
  double esec;
  struct timeval tp;
  struct timezone tzp;

  gettimeofday(&tp, &tzp);

  if ( zsec == 0 ) zsec = tp.tv_sec;
  if ( zusec == 0 ) zusec = tp.tv_usec;

  *retval = (tp.tv_sec - zsec) + (tp.tv_usec
- zusec ) * 0.000001 ;
}

void hpcwall_(double *retval) {
hpcwall(retval); } /* Otra convención */
```

Dado que a menudo necesitará usted tanto el tiempo de CPU como la hora real, y que continuamente estará calculando la diferencia entre sucesivas llamadas a tales rutinas, puede que quiera escribir una rutina que retorne el tiempo de reloj real y el tiempo de CPU cada vez que sea llamada, como sigue:

```
      SUBROUTINE HPCTIM(WTIME,CTIME)
      IMPLICIT NONE
```

```
*
      REAL WTIME,CTIME
      COMMON/HPCTIMC/CBEGIN,WBEGIN
      REAL*8 CBEGIN,CEND,WBEGIN,WEND
      REAL ETIME,CSCRATCH(2)
*
      CALL HPCWALL(WEND)
      CEND=ETIME(CSCRATCH)
*
      WTIME = WEND - WBEGIN
      CTIME = CEND - CBEGIN
*
      WBEGIN = WEND
      CBEGIN = CEND
      END
```

Utilizando la Información de Cronometraje

Puede usted obtener mucha información de las facilidades de cronometraje que le proporciona una máquina UNIX. No sólo puede decir cuánto tiempo lleva realizar cierto trabajo, sino también obtener indicios que le digan si la máquina está operando eficientemente, o si hay algún problema que requiera ser solucionado, tal como una memoria inadecuada.

Una vez que el programa está ejecutándose con todas las anomalías antes explicadas, puede usted registrar el tiempo como una línea base. Si está afinándolo, tal línea base será una referencia con la cuál podrá usted afirmar si el proceso de afinación ha mejorado mucho (o poco) las cosas. Si está realizando un banco de pruebas, puede usar dicha línea base para juzgar cuánto incremento global de rendimiento le está dando una máquina nueva. Pero recuerde observar también las otras cifras - paginación, uso de CPU, etc. ya que pueden diferir de máquina en máquina por razones que no están correlacionadas llanamente con el rendimiento de la CPU. Usted quiere asegurarse de obtener la imagen global.

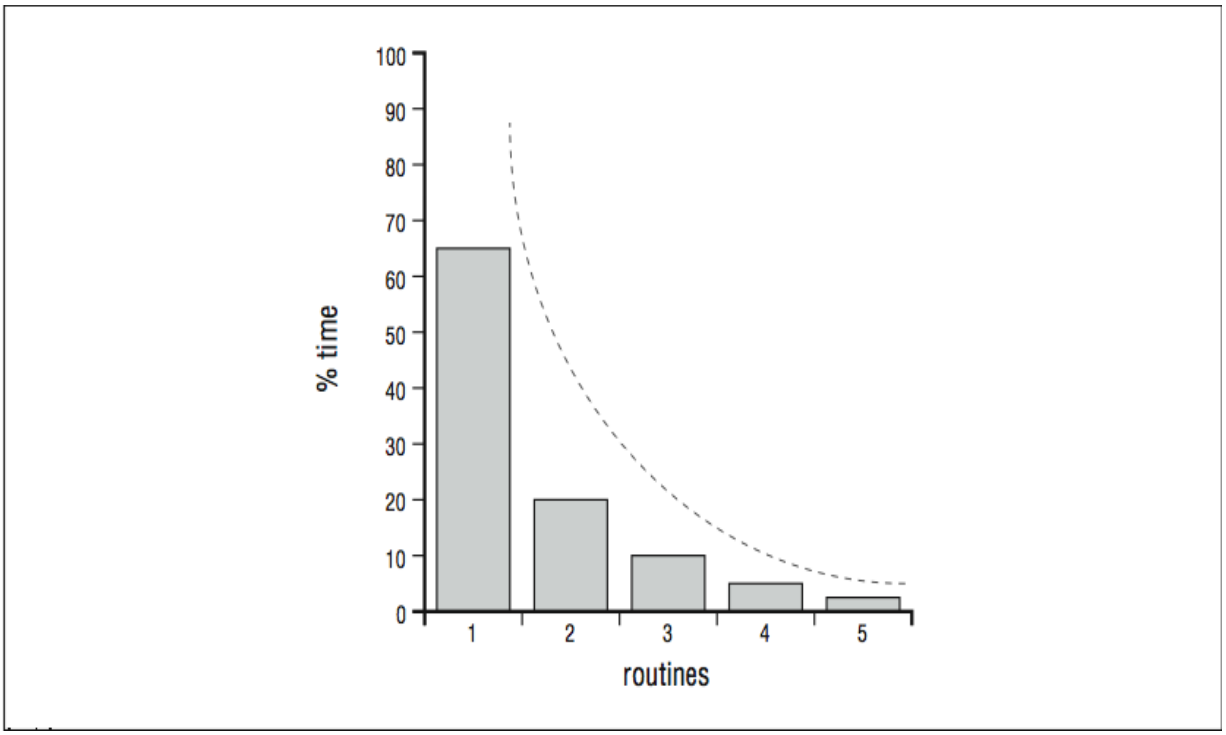
Cronometraje y Perfilado - Perfilado de Subrutinas

A veces va a desear más detalle que el que proporciona el cronometraje global de la aplicación, pero tal vez no tenga tiempo de modificar el código para insertar varios cientos de llamadas a *etime* en su código. Los perfiles también son muy útiles cuando le han entregado un extraño programa de 20,000 líneas de código, y le han pedido que averigüe cómo funciona y luego mejore su rendimiento.

La mayoría de los compiladores proporcionan la facilidad de insertar llamadas cronométricas automáticas en su código, tanto a la entrada como a la salida de cada rutina, a tiempo de compilación. Cuando se ejecute su programa, se registrarán los tiempos de entrada y salida, y luego se volcarán en un archivo. Una herramienta separada resume los patrones de ejecución y produce un reporte que muestra el porcentaje del tiempo gastado en cada una de las rutinas que usted escribió, así como en las rutinas de biblioteca.

El perfil debe proporcionarle a usted un sentido de la forma en que se está ejecutando. Esto es, puede ver que el 10% del tiempo se gasta en la subrutina A, el 5% en la subrutina B, etc. Naturalmente, si junta todas las subrutinas deben sumar el 100% del tiempo de ejecución global. A partir de estos porcentajes puede construir usted una imagen - un *perfil* — de la distribución de la ejecución cuando el programa corre. Aunque no sean representativos de alguna herramienta de perfilado en particular, los histogramas en [link] y [link] representan estos porcentajes, ordenados de izquierda a derecha, donde cada columna vertical representa una rutina diferente. Ayudan a ilustrar las diferentes formas de perfil.

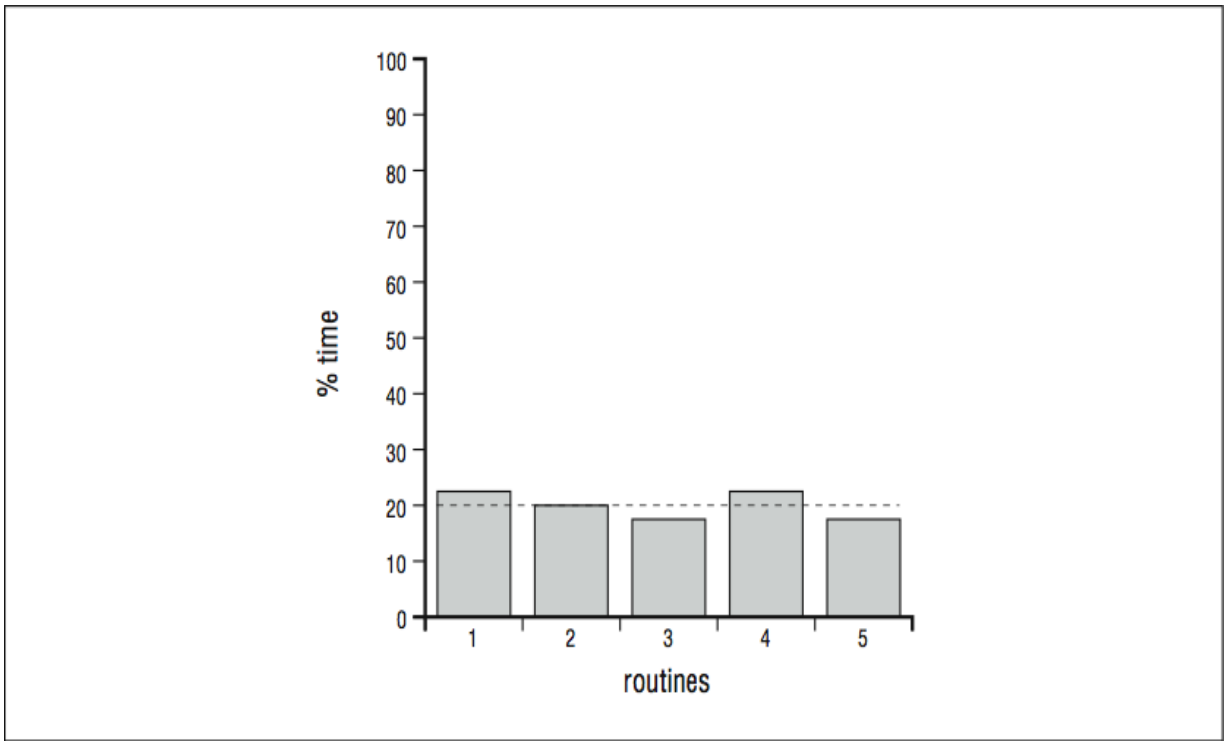
Perfil Agudo - Dominado por la Rutina 1



Un *perfil agudo* nos dice que la mayoría del tiempo se está gastando en uno o dos procedimientos, y que si quiere usted mejorar el rendimiento del programa, debe enfocar sus esfuerzos en afinar dichos procedimientos. Una optimización menor en una línea de código que se ejecuta intensamente puede a veces tener un enorme efecto en el tiempo de ejecución global, dando la oportunidad correcta. Un *perfil plano*,[footnote] por otra parte, le indica que el tiempo de ejecución se reparte entre muchas rutinas, y el esfuerzo gastado en optimizar cualquiera de ellas será poco benéfico para acelerar el programa. Por supuesto, también hay programas cuyo perfil de ejecución cae en algún punto intermedio.

A menudo e abusa un poco del término "perfil plano". Lo estamos usando para describir un perfil que muestra una distribución homogénea de tiempo a lo largo del programa. También verá que se emplea para marcar una diferencia de un perfil de un perfil del grafo de llamadas, como se describe más abajo.

Perfil plano - ninguna rutina predomina



No podemos predecir con absoluta certeza qué será lo que encuentre usted cuando perfile sus programas, pero hay algunas tendencias generales. Por ejemplo, los códigos científicos e ingenieriles construidos alrededor de soluciones matriciales a menudo exhiben perfiles muy agudos. El tiempo de ejecución está dominado por el trabajo realizado en un puñado de rutinas. Para afinar el código, necesita enfocar sus esfuerzos en tales rutinas, hasta hacerlas más eficientes. Puede que ello involucre reestructurar ciclos para exponer el paralelismo, proporcionar indicios al compilador, o reacomodar las referencias a memoria. En cualquier caso, el reto es tangible; puede ver los problemas que debe solucionar.

Por supuesto, existen límites a la mejora en tiempo de ejecución que obtendrá afinando una o dos rutinas. Una regla de oro citada frecuentemente es la *Ley de Amdahl*, obtenida a partir de las observaciones realizadas en 1967 por uno de los diseñadores de la serie 360 de IBM y fundador de Amdahl Computer, Gene Amdahl. Estrictamente hablando, sus observaciones fueron acerca del potencial de rendimiento de las computadoras paralelas, pero la gente ha adaptado la Ley de Amdahl para describir otras cosas por igual. Para nuestros propósitos, se puede citar así: digamos que tiene un programa con dos partes, una que puede optimizarse

hasta hacerla infinitamente rápida, y otra que no puede optimizarse en absoluto. Incluso si la porción optimizable supone como mucho el 50% del tiempo de ejecución inicial, cuando mucho será usted capaz de recortar el tiempo de ejecución total a la mitad. Esto es, su tiempo de ejecución estará dominado eventualmente por la porción que no puede optimizarse. Esto pone un límite máximo a sus expectativas de afinación.

Incluso a pesar del limitado retorno de inversión que sugiere la Ley de Amdahl, afinar un programa con un perfil agudo puede dar sus recompensas. Los programas con perfiles planos son mucho más difíciles de afinar. A menudo se trata de códigos de sistema, aplicaciones no numéricas y variedades de códigos numéricos sin soluciones matriciales. Se requiere de un enfoque de afinación global para reducir, a un nivel justificable, el tiempo de ejecución de un programa con un perfil plano. Por ejemplo, a veces puede usted optimizar el uso de la cache de instrucciones, lo cuál resulta complicado por la distribución equitativa de actividades del programa entre un gran número de rutinas. También puede ayudar a reducir la sobrecarga del llamado a subrutinas, plegando los invocados dentro de los invocadores. Ocasionalmente, puede encontrar un problema de referencia a memoria que es endémico al programa completo - y uno que puede arreglarse de un solo golpe.

Cuando estudie un perfil, puede que encuentre un porcentaje inusualmente grande de tiempo gastado en rutinas de biblioteca, tales como `log`, `exp` o `sin`. A menudo tales funciones se están realizando en rutinas de software, en vez de en línea. Puede reescribir su código para eliminar algunas de estas operaciones. Otro patrón importante que debe buscarse es aquel en el cual una rutina consume mucho más tiempo del que usted espera. Un tiempo de ejecución inesperado puede indicar que está accediendo a la memoria en un patrón que resulta malo desde el punto de vista del rendimiento, o que algún aspecto del código no puede optimizarse apropiadamente.

En cualquier caso, para obtener un perfil se requiere de una herramienta de perfilado. Uno o dos *perfiladores de subrutinas* vienen de forma estándar con los ambientes de desarrollo de software de todas las máquinas UNIX. Discutiremos dos de ellos: *prof* y *gprof*. Además, mencionaremos unos pocos perfiladores que actúan línea por línea. Los perfiladores de subrutina

pueden proporcionarle a usted una vista global general acerca de dónde está consumiéndose el tiempo. Probablemente deba comenzar con *prof*, si lo tiene (lo cuál es común en muchas máquinas). De otra forma, use *gprof*. Después de eso, puede moverse a un perfilador de línea por línea, si requiere conocer cuáles sentencias requieren de más tiempo.

prof

prof es la herramienta de perfilado más común en UNIX. En cierto sentido, es una extensión del compilador, enlazador y de las bibliotecas de código objeto, más unas pocas utilidades extras, de forma que es difícil hallar una sola cosa y decir "esto es lo que perfila su código." *prof* trabaja muestreando periódicamente el contador de programa conforme se ejecuta su aplicación. Para permitir el perfilado, usted debe recompilarla y re-enlazarla usando la bandera `-p`. Por ejemplo, si su programa tiene dos módulos, *stuff.c* y *junk.c*, requiere compilar y enlazar de acuerdo al código siguiente:

```
% cc stuff.c -p -O -c
% cc junk.c -p -O -c
% cc stuff.o junk.o -p -o stuff
```

Esto crea un binario *stuff* que está listo para el perfilado. Usted no requiere hacer nada especial para ejecutarlo. Sólo trátelo normalmente, tecleando `stuff`. Como se están recolectando las estadísticas a tiempo de ejecución, toma un poco más de lo usual en ejecutarse. [footnote] Una vez completado, habrá un nuevo archivo llamado *mon.out* en el directorio donde lo ejecutó. Este archivo contiene la historia de *stuff* en forma binaria, así que no puede observarla directamente. Use la utilidad *prof* para leer *mon.out* y crear un perfil de *stuff*. Por defecto, la información se escribe a su pantalla porque es la salida estándar, aunque puede redirigirla fácilmente a un archivo:

Recuerde: el código con el perfilado activo toma más en ejecutarse. Debe recompilar y re-enlazar el programa completo *sin* la bandera `-p` cuando

haya finalizado el perfilado.

```
% prof stuff > stuff.prof
```

Para poder explorar cómo funciona el comando *prof*, hemos creado la siguiente aplicación, *bucles.c*, que es pequeña y ridícula. Contiene una rutina principal y tres subrutinas para las cuáles puede usted predecir la distribución de tiempo, con sólo observar el código.

```
main () {
    int l;
    for (l=0;l<1000;l++) {
        if (l == 2*(l/2)) foo ();
        bar();
        baz();
    }
}
foo (){
    int j;
    for (j=0;j<200;j++)
}
bar () {
    int i;
    for (i=0;i<200;i++);
}
baz () {
    int k;
    for (k=0;k<300;k++);
}
```

Nuevamente, debe compilar y enlazar *bucles* con la bandera `-p`, ejecutar el programa, y luego ejecutar la utilidad *prof* para extraer un perfil, tal como

sigue:

```
% cc bucles.c -p -o bucles
% ./bucles
% prof bucles > bucles.prof
```

El siguiente ejemplo muestra cómo debe verse *bucles.prof*. Hay seis columnas.

```
%Time Seconds Cumsecs #Calls msec/call Name
 56.8    0.50    0.50   1000    0.500  _baz
 27.3    0.24    0.74   1000    0.240  _bar
 15.9    0.14    0.88    500    0.28   _foo
  0.0    0.00    0.88      1    0.
_creat
  0.0    0.00    0.88      2    0.
_profil
  0.0    0.00    0.88      1    0.     _main
  0.0    0.00    0.88      3    0.
_getenv
  0.0    0.00    0.88      1    0.
_strcpy
  0.0    0.00    0.88      1    0.
_write
```

Las columnas pueden describirse como sigue:

- `%Time` Porcentaje de tiempo de CPU consumido por esta rutina
- `Seconds` Tiempo de CPU consumido por esta rutina
- `Cumsecs` Un acumulado del tiempo consumida por esta rutina y todas las que la preceden en la lista
- `Calls` El número de veces que fue llamada esta rutina en particular

- `msec/call` Segundos divididos entre el número de llamadas, dando el promedio de tiempo tomado por cada invocación de la rutina
- `Name` El nombre de esta rutina

Las tres rutinas superiores listadas son del propio *bucles.c*. Puede observar una entrada que representa a la rutina principal (main), ubicada en la segunda mitad de la lista. Dependiendo del vendedor, los nombres de las rutinas pueden contener caracteres de guión bajo como prefijo o sufijo, y siempre habrá listadas algunas rutinas que usted no reconozca. Son contribuciones de la biblioteca C y posiblemente de las bibliotecas de FORTRAN, si está usando este lenguaje. El perfilado también introduce algo de sobrecarga en la ejecución, que a menudo se muestra como una o dos subrutinas en la salida de *prof*. En este caso, la entrada llamada `_profil` representa el código insertado por el enlazador para recolectar los datos de perfilado a tiempo de ejecución.

Si fuera nuestra intención afinar *bucles*, deberíamos considerar que un perfil como el mostrado en la figura superior es un buen signo. La primera rutina en la lista toma 50% del tiempo, así que cuando menos hay una posibilidad de que hagamos algo con ella que pueda tener un impacto significativo en el rendimiento global. (Por supuesto, con un programa tan trivial como *bucles*, hay mucho que podemos hacer, puesto que *bucles* no hace nada.)

gprof

Así como es importante conocer cómo se distribuye el tiempo durante la ejecución de su programa, también es valioso ser capaz de decir quién invocó a quién en la lista de rutinas. Imagine, por ejemplo, si algo etiquetado como `_exp` aparece posicionado alto en la lista de salida de prof. Usted puede decir: "Mmmm, no recuerdo haber invocado a nada llamado `exp()`. ¡No tengo ni idea de dónde viene!” Un árbol de llamados a subrutinas le ayudará a encontrarlo.

Puede pensarse en las subrutinas y funciones como miembros de un árbol familiar. En la parte más alta del árbol, o raíz, está una rutina que precede a la rutina principal que usted codificó en su aplicación; es quien llama a su rutina principal, que a su vez llama a otras, y así sucesivamente,

descendiendo hasta los nodos hoja del árbol. El nombre apropiado para este árbol es un *grafo de llamadas*.[footnote] La relación entre rutinas y nodos en el grafo es uno de padres e hijos. Nos referimos a los nodos separados por más de un salto como ancestros y descendientes.
No tiene por qué ser un árbol. Cualquier subrutina puede tener más de un padre. Es más, las rutinas recursivas introducen ciclos en el grafo, durante los cuales los hijos llaman a uno de sus padres.

La Figura 6-4 muestra gráficamente la clase de grafo de llamadas que puede verse en una aplicación pequeña. `main` es el padre o ancestro de la mayoría de las demás rutinas. `G` tiene dos padres, `E` y `C`. Otra rutina, `A`, no parece tener ningún ancestro o descendiente. Este problema puede suceder cuando no se compilan las rutinas con el perfilado activo, o cuando no han sido invocadas - tal como pudiera ser el caso si `A` fuera un manejador de excepciones.

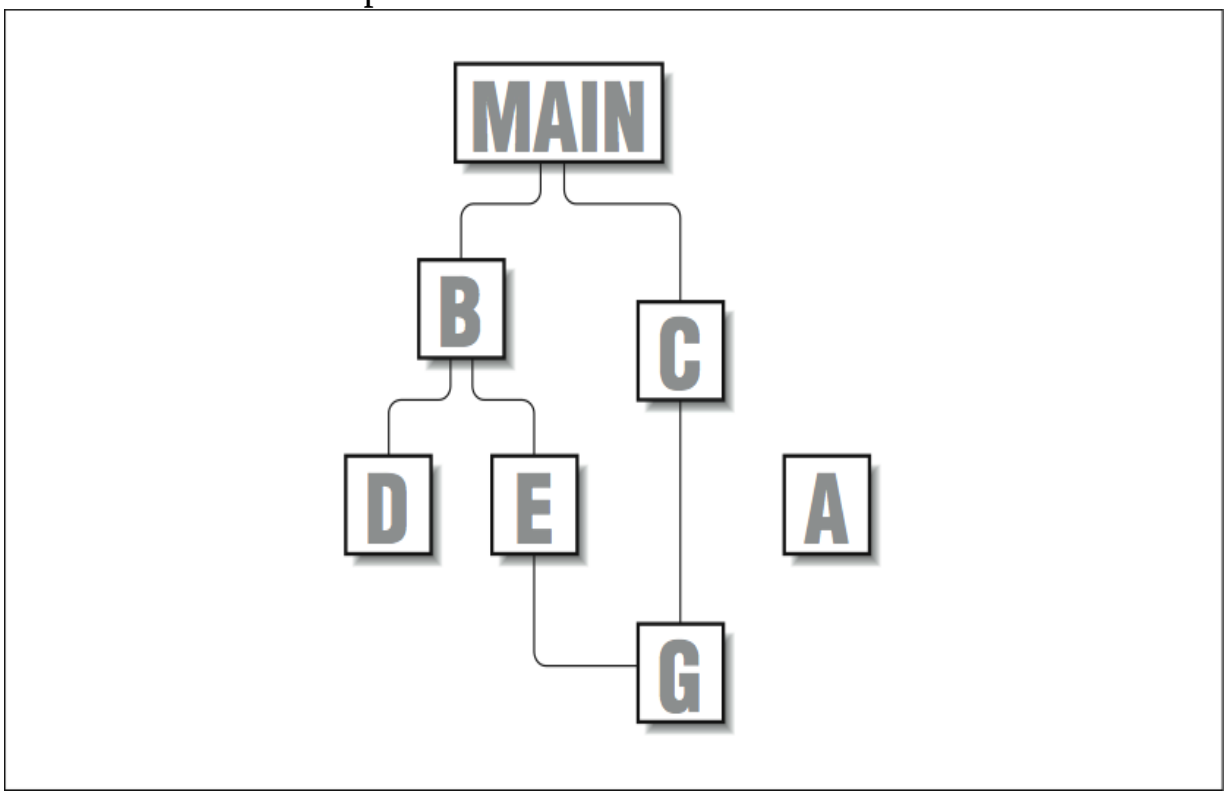
El software de perfilado de UNIX que puede extraer esta clase de información se llama *gprof*. Replica las habilidades de *prof*, además de proporcionar un perfil de grafo de llamados, de forma que pueda usted observar quién llama a quién, y cuán frecuentemente. El perfil del grafo de llamadas es útil si está tratando de averiguar cómo trabaja un fragmento de código, o de dónde proviene una rutina desconocida, o si está buscando candidatos para hacer una subrutina en línea.

Para usar el perfilado del grafo de llamados, usted debe seguir los mismos pasos que con *prof*, excepto que la bandera `-pg` se sustituye por la bandera `-p`.[footnote] Adicionalmente, cuando llega el momento de producir el perfil, debe usted usar el programa de utilidad *gprof* en vez de *prof*. Una diferencia más es que el nombre del archivo de estadísticas es *gmon.out* en vez de *mon.out*:
En las máquinas HP, la bandera es `-G`.

```
% cc -pg stuff.c -c
% cc stuff.o -pg -o stuff
% stuff
```

```
% gprof stuff > stuff.gprof
```

Grafo de llamadas simple



La salida de `gprof` se divide en tres secciones:

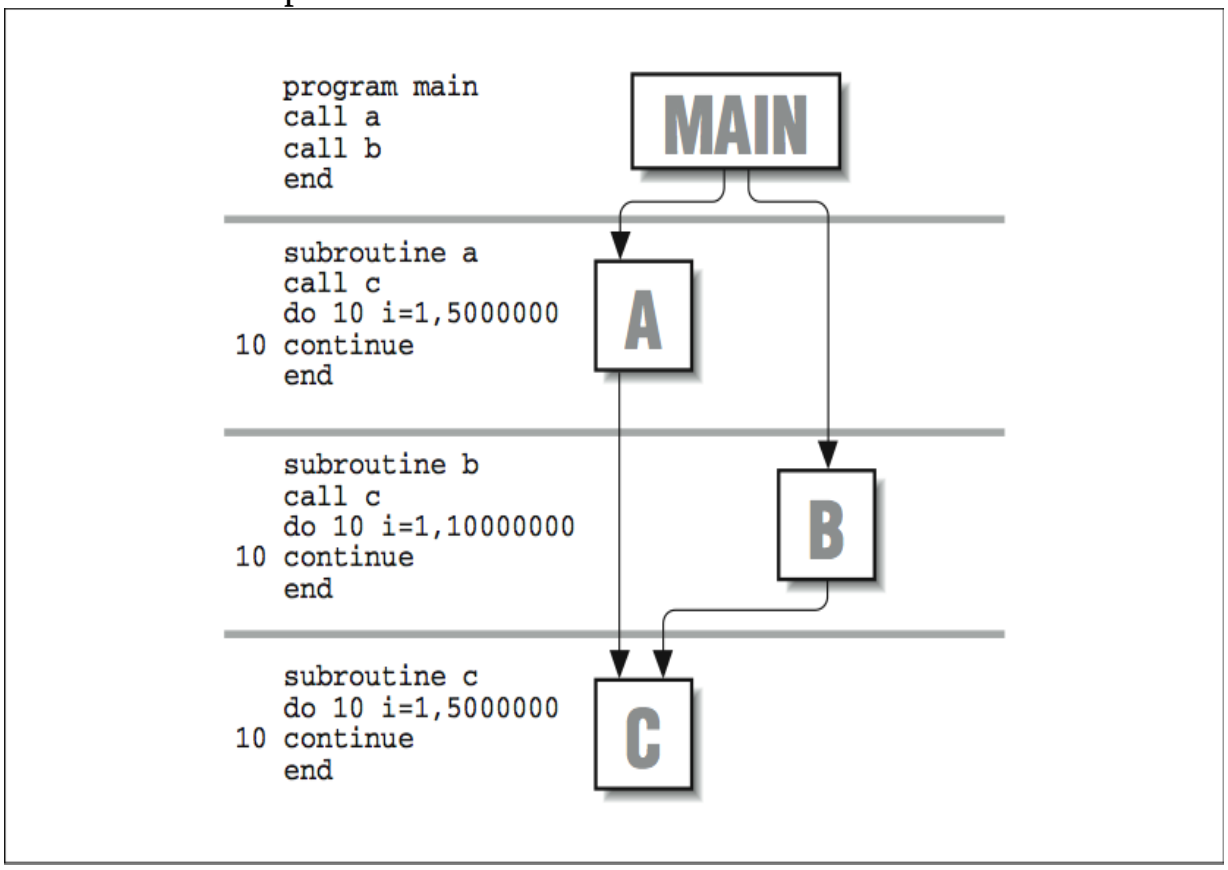
- Perfil del grafo de llamadas
- Perfil de cronometraje
- Índice

La primera sección es textualmente el mapa del grafo de llamadas. La segunda lista las subrutinas, el porcentaje de tiempo dedicado a cada una, el número de llamadas, etc. (similar a *prof*). La tercera sección es una referencia cruzada, de forma que pueda usted localizar las rutinas por número, en vez de por nombre. Esta sección resulta especialmente útil en aplicaciones grandes, porque las rutinas se ordenan basadas en la cantidad de tiempo que emplean, y puede resultar difícil localizar una en particular buscando su nombre. Inventemos otra aplicación trivial para ilustrar cómo funciona *gprof*. [link] muestra una pequeña pieza de código FORTRAN,

junto con un diagrama que indica cómo están conectadas las rutinas. Tanto la subrutina `A` como la `B` son invocadas por `MAIN`, y, a su vez, cada una llama a `C`. El siguiente ejemplo muestra una sección de la salida del perfil del grafo de llamadas de *gprof*:[footnote]

Con el fin de no consumir demasiado espacio, recortamos la sección más relevante para nuestra discusión y la incluimos en este ejemplo. Hay mucho más, incluyendo llamados a rutinas de configuración y sistema, del tipo de las que ve cuando ejecuta *gprof*.

FORTRAN example




```
called/total       parents
      index  %time     self descendants
called+self    name        index

called/total       children
```

```
        ....                        ....
....

                        0.00          8.08      1/1
_main [2]
      [3]    99.9       0.00          8.08      1
_MAIN_ [3]
                        3.23          1.62      1/1
_b_ [4]
                        1.62          1.62      1/1
_a_ [5]

      -----------------------------------------------
---

                        3.23          1.62      1/1
_MAIN_ [3]
      [4]    59.9       3.23          1.62      1
_b_ [4]
                        1.62          0.00      1/2
_c_ [6]

      -----------------------------------------------
---

                        1.62          1.62      1/1
_MAIN_ [3]
      [5]    40.0       1.62          1.62      1
_a_ [5]
                        1.62          0.00      1/2
_c_ [6]

      -----------------------------------------------
---

                        1.62          0.00      1/2
```

```
_a_ [5]
                                1.62            0.00        1/2
_b_ [4]
      [6]     39.9        3.23            0.00        2
_c_ [6]
```

Emparedada entre cada conjunto de líneas punteadas está la información que describe una rutina dada y su relación respecto a padres e hijos. Es fácil decir qué rutina representa cada bloque, porque el mismo está desplazado más a la izquierda que los otros. Los padres se listan arriba, los hijos abajo. Como sucedió con `prof`, se agregaron guiones bajos a las etiquetas. [footnote] A continuación se muestra una descripción de cada una de las columnas:

Puede que haya observado que hay dos rutinas principales: `_MAIN_` y `_main`. En un programa FORTRAN, `_MAIN_` es la verdadera rutina principal, que es invocada como una subrutina por `_main`, que proporciona una biblioteca del sistema a tiempo de enlace. Cuando perfila usted código en C, no verá `_MAIN_`.

- `index` Observará que cada nombre de rutina tiene asociado un número entre corchetes (`[n]`). Se trata de una referencia cruzada para ubicar rutina en cualquier parte del perfil. Si, por ejemplo, está usted buscando en el bloque que describe `_MAIN_` y quiere saber más acerca de sus hijos, digamos `_a_`, puede encontrarlo recorriendo hacia abajo la parte izquierda de la página en busca de su índice, `[5]`.
- `%time` El significado del campo `%time` es un poco diferente del que tiene para *prof*. En este caso describe el porcentaje de tiempo gastado en esta rutina *mas* el tiempo gastado en todos sus hijos. Le proporciona una forma rápida de determinar dónde encontrar las partes más ocupadas del grafo de llamadas.
- `self` Listado en segundos, la columna `self` tiene diferentes significados para los padres, la rutina en cuestión y sus hijos. Comenzando con la entrada central -la rutina misma- el valor `self` muestra cuánto tiempo global se dedicó a la rutina. En el caso de `_b_`, por ejemplo, esta cantidad fue de 3.23 segundos.
  Cada entrada en la columna `self` muestra la cantidad de tiempo que

puede atribuirse a llamadas desde los padres. Si observa la rutina `_c_`, por ejemplo, verá que consumió un tiempo total de 3.23 segundos. Pero note que tuvo dos padres: 1.62 segundos del tiempo se pueden atribuir a llamadas provenientes de `_a_`, y 1.62 segundos a las de `_b_`.

Para el hijo, la cantidad `self` muestra cuánto tiempo se gastó ejecutando cada hijo, debido a llamadas provenientes de esta rutina. Los hijos pueden haber consumido más tiempo global, pero el único tiempo contabilizado es aquél atribuíble a llamadas desde esta subrutina. Por ejemplo, `_c_` acumuló 3.23 segundos globales, pero si observa en el bloque describiendo `_b_`, verá a `_c_` listada como un hijo con sólo 1.62 segundos. Este fue el tiempo total gastado ejecutando `_c_` del lado de `_b_`.

- `descendants` Como ocurren con la columna `self`, los vaores en la columna de descendientes tienen diferentes significados para la rutina, sus padres y los hijos. Para la rutina misma, muestra el número de segundos gastados en todos sus descendientes.

  Para los padres de la rutina, esta columna describe cuánto tiempo gastado por la rutina puede trazarse a partir de llamadas realizadas por cada padre. Observando nuevamente la rutina `_c_`, puede observar que de su tiempo total, 3.23 segundos, 1.62 segundos son atribuíbles a cada uno de sus padres, `_a_` y `_b_`.

  Para los hijos, la columna de descendientes muestra cuánto del tiempo de los hijos puede atribuirse a llamadas realizadas desde esta rutina. El hijo puede haber acumulado más tiempo global, pero aquí sólo se despliega el tiempo asociado con llamadas desde esta rutina.
- `calls` La columna `calls` muestra el número de veces que se invocó cada rutina, así como la distribución de dichas llamadas asociadas tanto con padres como con hijos. Comenzando con la rutina misma, la cantidad en la columna `calls` muestra el número total de entradas a la rutina. En situaciones donde la rutina se invoca a sí misma, también observará usted un +*n* agregado inmediatamente, mostrando que se realizaron *n* llamadas recursivas adicionales.

  Las cantidades de padres e hijos se expresan como tasas. Para los padres, la tasa *m/n* debe leerse como “de las *n* veces que se invocó la rutina, *m* vinieron de su padre.” Para el hijo, debe leerse como "de las *n* veces que fue invocado este hijo, *m* provinieron de esta rutina.”

Perfil Plano de gprof

Como mencionamos previamente, *gprof* también produce un perfil de cronometraje (también conocido como un perfil "plano", un término algo confuso) similar al que produce *prof*. Unos pocos campos son diferentes de *prof*, y también hay algo de información extra, así que será de ayuda explicarlo brevemente. El siguiente ejemplo muestra unas pocas de las primeras líneas de un perfil plano de *gprof* para el programa *stuff*. Reconocerá del programa original las tres rutinas superiores. Las otras son funciones de biblioteca incluidas a tiempo de enlace.

```
      %    cumulative     self              self
total
     time    seconds     seconds    calls  ms/call
ms/call  name
     39.9       3.23        3.23        2  1615.07
1615.07  _c_ [6]
     39.9       6.46        3.23        1  3230.14
4845.20  _b_ [4]
     20.0       8.08        1.62        1  1620.07
3235.14  _a_ [5]
      0.1       8.09        0.01        3     3.33
3.33  _ioctl [9]
      0.0       8.09        0.00       64     0.00
0.00  .rem [12]
      0.0       8.09        0.00       64     0.00
0.00  _f_clos [177]
      0.0       8.09        0.00       20     0.00
0.00  _sigblock [178]
      ...        ....       ....        .         .
.       ......
```

He aquí el significado de cada columna:

- `%time` Nuevamente, vemos un campo que describe el tiempo de ejecución de cada rutina, como un porcentaje del tiempo global del programa. Como puede usted esperar, todas las entradas esta columna deben totalizar (aproximadamente) un 100%.
- `cumulative` seconds. Para una rutina dada, la columna llamada "segundos acumulados" hace un recuento de la suma del tiempo de ejecución ejecución tomado por todas las rutinas precedentes, más el tiempo propio. Conforme la vaya revisando hacia abajo, verá que los números se aproximan asintóticamente al tiempo total del programa.
- `self` seconds La contribución individual de cada rutina al tiempo total de ejecución.
- `calls` El número de veces que fue invocada esta rutina en particular.
- `self ms/call` Los segundos gastados adentro de la rutina, dividido entre el número de llamadas. Ello nos da una duración promedio del tiempo tomado por cada invocación a la rutina. La cantidad está medida en milisegundos.
- `total ms/call` Los segundos gastados adentro de la rutina más sus descendientes, dividido entre el número de llamados.
- `name` El nombre de la rutina. Observe que de nuevo aparece aquí el número de referencia cruzada.

Acumulando los Resultados de Varias Corridas de gprof

Es posible acumular estadísticas provenientes de múltiples corridas, de forma que obtenga usted una instantánea del comportamiento del programa con una variedad de conjuntos de datos. Por ejemplo, digamos que quiere usted perfilar una aplicación - llamémosla *bar* — con tres conjuntos de datos distintos. Puede realizar las corridas separadamente, guardando los archivos *gmon.out* conforme las ejecuta, y luego combinar los resultados en un solo perfil al terminar:

```
% f77 -pg bar.f -o bar
% bar < data1.input
% mv gmon.out gmon.1
% bar < data2.input
```

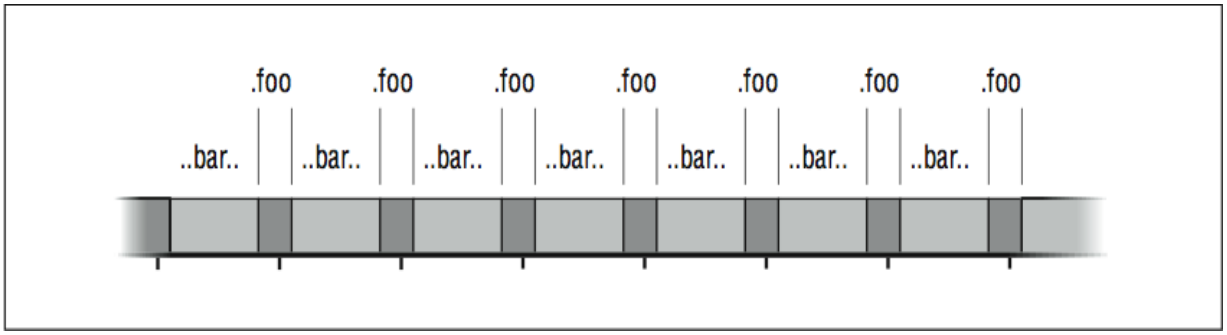
```
% mv gmon.out gmon.2
% bar < data3.input
% gprof bar -s gmon.1 gmon.2 gmon.out >
gprof.summary.out
```

En el perfil de ejemplo, cada corrida crea un nuevo archivo *gmon.out* que luego renombramos para que no sea encimado por el siguiente. Al final, *gprof* combina la información de cada uno de los archivos de datos para producir un perfil sumario de *bar* en el archivo *gprof.summary.out*. Adicionalmente (aunque no se muestra aquí), *gprof* crea un archivo llamado *gmon.sum* que contiene los datos mezclados a partir de los tres archivos de datos originales. *gmon.sum* tiene el mismo formato que *gmon.out*, de forma que pueda usted usarlo como entrada para producir otros perfiles mezclados si así lo requiere.

Al menos en forma, la salida del perfil mezclado luce exactamente igual que el de una corrida individual. Pero hay un par de cosas interesantes que debemos señalar. Por un lado, la rutina `main` parece haber sido invocada más de una vez - de hecho, una vez por cada corrida. Además, dependiendo de la aplicación, las múltiples ejecuciones tienden o bien a suavizar el contorno del perfil, o bien a exagerar sus características. Puede imaginar cómo es que esto sucede. Si se está invocando constantemente una única rutina, mientras que las otras vienen y van conforme cambian los datos de entrada, ésta toma una importancia creciente en sus esfuerzos de afinación.

Unas Pocas Palabras Acerca de la Exactitud

En aquellos procesadores que se ejecutan a 600 MHz o más, el tiempo que transcurre entre muestras de 60 Hz y 100 Hz es una verdadera eternidad. Es más, puede que experimente usted errores de cuantificación cuando la frecuencia de muestreo es fija, como es el caso en muestras de 1/100 y 1/60 de segundo. Para usar un ejemplo exagerado, asumamos que la línea de tiempo en [link] muestra invocaciones alternadas a dos subrutinas, `BAR` y `FOO`. Las marcas cronométricas representan los puntos de muestreo del perfilado.
Errores de cuantificación al perfilar



BAR y FOO se ejecutan por turnos. Pero BAR toma más tiempo que FOO. Y dado que el intervalo de muestreo es muy cercano a la frecuencia en que ambas se alternan, tenemos un error de cuantificación: la mayoría de las muestras se están tomando mientras FOO se está ejecutando. Así, el perfil nos dice que FOO usa más tiempo de CPU que BAR.

Hemos descrito y probado los perfiladores de subrutinas reales que han estado disponibles en UNIX durante años. En muchos casos, los vendedores tienen herramientas mucho mejores, ya sea a la venta o gratis. Si usted está haciendo un trabajo de afinación serio, pregunte a su representante de ventas qué otras herramientas le ofrece.

Cronometraje y Perfilado - Perfiladores de Bloques Básicos

Existen varias buenas razones para desear un nivel de detalle más fine que el que puede obtener usando un perfilador a nivel de subrutina. Para los humanos que tratan de entender cómo se usa una subrutina o función, un perfilador que indique cuáles líneas de código fuente se están ejecutando realmente, y cuán a menudo, resulta invaluable; unas pocas pistas acerca de dónde enfocar sus esfuerzos de afinación pueden ahorrarle mucho tiempo. Además, tal tipo de perfilador le ahora el descubrimiento de que una optimización particularmente inteligente no resulta en diferencia alguna, porque la colocó en una sección cuyo código nunca se ejecuta.

Como parte de una estrategia global, un perfilador de subrutinas puede dirigirlo hacia un puñado de rutinas que contabilizan la mayor parte del tiempo de ejecución, pero se necesita de un *perfilador de bloques básicos[footnote]* para que pueda usted obtener las líneas de código asociadas.
Un bloque básico es una sección de código con un único punto de entrada y un único punto de salida. Si sabe cuántas veces se entró al bloque, sabe cuántas veces se ejecutó cada sentencia en el bloque, lo cuál le está dando un perfil línea por línea. El concepto de bloque básico se explica con más detalle en [link]

Los perfiladores de bloques básicos también proporcionan a los compiladores la información que necesitan para realizar sus propias optimizaciones. Muchos compiladores trabajan a ciegas: pueden reestructurar y desenrollar ciclos, pero no pueden decir cuándo merece la pena hacerlo. Y lo que es todavía peor, a veces las optimizaciones mal ubicadas ¡tienen el efecto adverso de volver más lento el código! Ello puede deberse al gravamen a que se ve sometida la cache de instrucciones, a las pruebas innecesarias introducidas por el compilador, o a hipótesis incorrectas acerca de qué camino tomará una bifurcación a tiempo de ejecución. Si el compilador puede interpretar automáticamente los resultados de un perfilador de bloques básicos, o si puede usted proporcionarle algunos indicios, a menudo significa un tiempo de ejecución reducido con poco esfuerzo de su parte.

Existen muchísimos perfiladores de bloques básicos en el Mundo. Lo más parecido a un estándar, *tcov*, se distribuye con las estaciones de trabajo Sun. Es estándar porque la base instalada es muy grande. En las estaciones de trabajo basadas en MIPS, tales como las Silicon Graphics y las DEC, el perfilador (empaquetado como una extensión de *prof*) se llama *pixie*. Explicaremos brevemente cómo ejecutar cada perfilador, usando un conjunto razonable de opciones. Puede consultar sus respectivas páginas de manual en busca de otras opciones.

tcov

tcov, disponible para estaciones de trabajo Sun y otras máquinas SPARC que ejecuten SunOS, proporciona estadísticas de ejecución que describen el número de veces que fue ejecutada cada sentencia del código fuente. Es muy fácil de usar. Para ilustrarlo, asumamos que tenemos un programa fuente llamado *foo.c*. Los siguientes pasos crean un perfil de bloques básicos:

```
% cc -a foo.c -o foo
% foo
% tcov foo.c
```

La opción `-a` le indica al compilador que incluya el soporte necesario para *tcov*.[footnote] Se crean varios archivos durante el proceso. Uno llamado *foo.d* acumula una historia de las frecuencias de ejecución adentro del programa *foo*. Esto es, los datos antiguos se actualizan con datos nuevos cada vez que se ejecuta *foo*, así que puede usted obtener una imagen panorámica de qué sucede adentro de *foo*, dada una variedad de conjuntos de datos. Sólo recuerde limpiar los datos antiguos se quiere comenzar desde cero. El perfil en sí se guarda en un archivo llamado *foo.tcov*.
En los sistemas Solaris de Sun se usa a opción `-xa`.

Veamos una ilustración. A continuación está un pequeño programa en C que realiza una ordenación de 10 enteros mediante el algoritmo de la

burbuja:

```
      int n[] = {23,12,43,2,98,78,2,51,77,8};
      main ()
        {
          int i, j, ktemp;
          for (i=10; i>0; i--) {
            for (j=0; j<i; j++) {
              if (n[j] < n[j+1]) {
                ktemp = n[j+1], n[j+1] = n[j],
n[j] = ktemp;
              }
            }
          }
        }
```

tcov produce un perfil de bloques básicos que contiene el conteo de la ejecución de cada línea del código fuente, además de algunas estadísticas de resumen (que no se muestran):

```
          int n[] = {23,12,43,2,98,78,2,51,77,8};
          main ()
      1 ->  {
                int i, j, ktemp;
     10 ->      for (i=10; i>0; i--) {
     10,      55 ->       for (j=0; j<i; j++) {
     55 ->         if (n[j] < n[j+1]) {
     23 ->           ktemp = n[j+1], n[j+1] =
n[j], n[j] = ktemp;
                   }
                 }
               }
```

```
1 ->   }
```

Los números a la izquierda le indican el número de veces que se entró a cada bloque. Por ejemplo, puede ver que a la rutina sólo se entró una vez, y que el conteo más alto ocurre en la condición `n[j] < n[j+1]`. *tcov* muestra más de un conteo sobre una línea en lugares donde el compilador ha creado más de un bloque.

pixie

pixie [footnote] es ligeramente diferente de *tcov*. En vez de reportar el número de veces que se ejecuta cada línea de código, pixie reporta el número de ciclos de reloj de máquina dedicados a cada línea. En teoría, puede usted usarlos para calcular la cantidad de tiempo gastada por sentencia, aunque no se representan anomalías como las fallas de cache. duendecillo, si se traduce literalmente. N. del T.

pixie funciona “pixificando” un archivo ejecutable que fue compilado y enlazado normalmente. A continuación mostramos la ejecución de *pixie* sobre el programa *foo* para crear un nuevo ejecutable llamado *foo.pixie*:

```
% cc foo.c -o foo
% pixie foo
% foo.pixie
% prof -pixie foo
```

También se creó un archivo llamado *foo.Addrs*, el cual contiene las direcciones de los bloques básicos dentro de *foo*. Cuando se ejecuta el nuevo programa, *foo.pixie*, crea un archivo llamado *foo.Counts* , que contiene la contabilidad de la ejecución de los bloques básicos cuyas direcciones están almacenadas en *foo.Addrs*. Los datos de *pixie* se acumulan de ejecución en ejecución. Las estadísticas se recuperan usando *prof* y la bandera especial **–pixie**.

La salida por defecto de *pixie* se divide en tres secciones, como sigue:

- Ciclos por rutina
- Conteo de la invocación de procedimientos
- Ciclos por línea básica

A continuación mostramos el listado de salida de la tercera sección para el programa de ordenación por el método de la burbuja:

```
      procedure (file)            line  bytes
cycles      %  cum %

      main (foo.c)                   7     44
605  12.11  12.11
      _cleanup (flsbuf.c)           59     20
500  10.01  22.13
      fclose (flsbuf.c)             81     20
500  10.01  32.14
      fclose (flsbuf.c)             94     20
500  10.01  42.15
      _cleanup (flsbuf.c)           54     20
500  10.01  52.16
      fclose (flsbuf.c)             76     16
400   8.01  60.17
      main (foo.c)                  10     24
298   5.97  66.14
      main (foo.c)                   8     36
207   4.14  70.28
      ....                          ..     ..
..    ...    ...
```

Aquí puede usted ver tres entradas para la rutina principal de *foo.c*, además de un número de rutinas de bibliotecas del sistema. Las entradas muestran el número de línea asociado y el número de ciclos de máquina dedicados a

ejecutar dicha línea, conforme se corrió el programa. Por ejemplo, la línea 7 de *foo.c* tomó 605 ciclos (12% del tiempo de ejecución).

Además de tener un impacto negativo en el rendimiento debido a las fallas de cache, el sistema de memoria virtual puede disminuir la velocidad de ejecución de su programa, si es demasiado grande para caber en la memoria del sistema, o está compitiendo con otros trabajos grandes por usar una memoria escasa.

En la mayoría de las implementaciones de UNIX, el sistema operativo descarga automáticamente al área de intercambio parte de las páginas de aquellos programas demasiado grandes para caber en la memoria disponible. No arrojan el programa completo; sólo sucede cuando la memoria está completamente justa, o cuando su programa ha estado inactivo durante un rato. En tal caso, algunas páginas individuales se ponen en el área de intercambio para su recuperación posterior. En primer término, debe usted estar consciente de que esto está sucediendo, si es que todavía no se ha percatado. Y en segundo término, si está pasando, los patrones de acceso a memoria se tornan críticos. Cuando las referencias sean muy dispersas, su tiempo de ejecución quedará dominado completamente por la E/S a disco.

Si lo planea con anticipación, puede hacer que un sistema de memoria virtual trabaje para usted cuando su programa sea demasiado grande para alojarse en la memoria física de la máquina. Las técnicas son exactamente las mismas que las empleadas para afinar una aplicación sin espacio en memoria principal administrada por software, o bucles anidados. Consiste en "bloquear" las referencias a memoria, de forma que los datos consumidos en los vecinos usen una mayor porción de cada página de memoria virtual antes de regresarla a disco para hacer espacio para otra. [footnote]
Examinamos las técnicas para bloqueos en el [link] Capítulo 8.

Calibrando el Tamaño de su Programa y el Tamaño de la Memoria

¿Cómo puede asegurar que se ha quedado sin espacio en memoria principal? Hay varias formas de comprobar la paginación en la máquina,

pero tal vez la prueba más sencilla es comparar el tamaño de su programa con la cantidad de memoria disponible. Esto se hace mediante el comando *size*:

```
% size myprogram
```

En una máquina con UNIX Sistema V, la salida será similar a la siguiente:

```
53872 + 53460 + 10010772 = 10118104
```

En un sistema derivado del UNIX de Berkeley, se verá similar a:

```
text      data      bss          hex
decimal
53872     53460     10010772     9a63d8
10118104
```

Los tres primeros campos describen la cantidad de memoria necesaria para tres porciones distintas de su programa. La primera, el segmento de texto, contabiliza las instrucciones de máquina que forman su programa. La segunda, el segmento de datos, incluye los valores inicializados en su programa, tales como los contenidos de las sentencias de datos, bloques comunes, externos, cadenas de caracteres, etc. El tercer componente, bss (block started by symbol), usualmente es el mayor. Describe un área de datos sin inicializar en su programa. Esta área está formada de bloques comunes que no están ocupados por un bloque de datos. El último campo es la suma de las tres secciones, en bytes.[footnote]
Advertencia: El comando *size* no le da una panorámica completa si su programa aparta memoria dinámicamente, o mantiene datos en la pila. Esta

área es especialmente importante para los programas en C y FORTRAN que crean arreglos grandes que no están en la sección COMMON.

Después, requiere saber cuánta memoria tiene su sistema. Desafortunadamente, no existe un comando UNIX estándar para eso. En los sistemas RS/6000, */etc/lscfg* se lo dice. En una máquina SGI, */etc/hinv* lo hace. Muchas implementaciones de UNIX Sistema V tienen un comando */etc/memsize*. En un derivado de Berkeley, puede teclear:

```
% ps aux
```

Este comando le devuelve un listado de todos los procesos ejecutándose en la máquina. Encuentre el proceso con el valor más grande en la columna `%MEM`. Divida el valor en el campo `RSS` entre el porcentaje de memoria usado, para obtener un valor aproximado de cuánta memoria tiene su máquina:

```
memory = RSS/(%MEM/100)
```

Por ejemplo, si el proceso más grande muestra un uso de memoria del 5% y un tamaño de conjunto residente (RSS, resident set size) de 840 KB, su máquina tiene 840000/(5/100) = 16 MB de memoria.[footnote] Si la respuesta del comando *size* muestra un total cercano a la cantidad de memoria que tiene, hay una buena posibilidad de que ocurra intercambio de páginas cuando lo ejecute - especialmente si está realizando otras cosas en la máquina al mismo tiempo.
¡También puede usted reiniciar su máquina! Ella le dirá cuánta memoria tiene disponible al arranque.

Comprobando los Fallos de Página

Sus herramientas de monitoreo de rendimiento del sistema le indican si su programa está realizando intercambio de páginas. Algo de intercambio está bien; los fallos de página ocurren de manera natural conforme se ejecuta el programa. Sin embargo, sea cuidadoso si está compitiendo por los recursos del sistema con otros usuarios. La imagen que obtendrá no será la misma que cuando tenga la computadora para usted solo.

Para comprobar la actividad de intercambio de páginas en un sistema UNIX derivado de Berkeley, use el comando *vmstat*. Comúnmente la gente lo invoca con un incremento de tiempo, de forma que reporte el intercambio de páginas a intervalos regulares:

```
% vmstat 5
```

Este comando produce una línea de salida cada cinco segundos:

```
      procs     memory              page
disk       faults     cpu
      r b w   avm   fre  re at  pi  po  fr  de  sr
s0 d1 d2 d3  in  sy  cs us sy id
      0 0 0   824 21568   0  0   0   0   0   0   0
0  0  0  0  20  37  13  0  1 98
      0 0 0   840 21508   0  0   0   0   0   0   0
1  0  0  0 251 186 156  0 10 90
      0 0 0   846 21460   0  0   0   0   0   0   0
2  0  0  0 248 149 152  1  9 89
      0 0 0   918 21444   0  0   0   0   0   0   0
4  0  0  0 258 143 152  2 10 89
```

Como puede observar, produce mucha información. Para nuestros propósitos, los campos importantes son `avm`, que significa "memoria virtual activa" (*active virtual memory*), `fre` o "memoria real libre" (*free real memory*), y los campos `pi` y `po`, cuyos números muestran la actividad de paginación. Cuando la cantidad en `fre` cae a valores cercanos a cero, y el

campo `po` muestra mucha actividad, es una indicación de que el sistema de memoria está saturándose.

En una máquina con SysV, la actividad de paginación puede observarse mediante el comando *sar*:

```
% sar -r 5 5
```

Este comando le muestra la cantidad de memoria libre y el espacio de intercambio disponible al instante. Si el valor de memoria libre está bajo, puede asumir que su programa realizará intercambio de páginas:

```
Sat Apr 18 20:42:19
[r] freemem freeswap
       4032    82144
```

Como mencionamos previamente, su desea ejecutar un trabajo mayor que el tamaño de memoria que posee su máquina, debe aplicar para la actividad de intercambio de páginas el mismo consejo que aplica para la actividad de cache.[footnote] Trate de minimizar el tamaño de incremento en los bucles de su código, y cuando no pueda, bloquear las referencias a memoria ayudará muchísimo.
Por cierto, ¿está recibiendo el mensaje “Out of memory?” Si está ejecutando *csh*, pruebe a teclear `unlimit` para ver si el mensaje desaparece. De otro modo, puede significar que no tiene suficiente espacio de intercambio disponible para ejecutar el trabajo.

Una nota sobre las herramientas de monitoreo de rendimiento de la memoria: debe comprobar con el vendedor de su estación de trabajo si tienen disponible alguna herramienta además de *vmstat* o *sar*. Puede que exista alguna herramienta mucho más sofisticada (y tal vez gráfica) que le ayude a comprender cómo es que su programa utiliza la memoria.

Cronometraje y Perfilado - Notas de Cierre

Hemos visto algunas de las herramientas para cronometraje y perfilado. Incluso aunque parezca que hemos cubierto mucho, hay otras clases de perfiladores que debiéramos cubrir - medidas de fallas de cache, análisis de dependencias a tiempo de ejecución, medición de operaciones de punto flotante por segundo, y varias más. Estos perfiles son buenos cuando está buscando ciertas anomalías en particular, tales como fallas de cache o uso de las líneas de espera de punto flotante. En algunas máquinas existen perfiladores para estas cantidades, pero no están ampliamente distribuidos.

Algo para tener en mente: cuando perfile código, a veces obtiene una visión muy limitada de la forma en que se usa el programa. Ello es especialmente cierto si puede realizar muchos tipos de análisis para muchos conjuntos distintos de datos de entrada. Trabajar con sólo uno o dos perfiles puede darle una vista distorsionada de cómo opera el código en su conjunto. Imagine el siguiente escenario: alguien le invita a dar su primer paseo en automóvil. Está sentado en el asiento del pasajero con un cuaderno y una pluma, y registra todo lo que sucede. Sus observaciones incluyen cosas tales como:

- El radio siempre está encendido.
- Los limpiaparabrisas nunca se usan.
- El automóvil sólo se mueve hacia adelante.

El peligro es que, dado lo limitado de su óptica acerca la forma en que se opera el auto, puede que quiera desconectar la perilla de encendido y apagado del radio, quitar los limpiaparabrisas y eliminar el engranaje de reversa. ¡Ello se convertirá en una verdadera sorpresa para la siguiente persona que intente estacionar en reversa el auto en un día lluvioso! El punto es que a menos que sea usted cuidadoso al recolectar datos de *todos los tipos* de usos posibles, puede que no tenga una panorámica real de cómo opera el programa. Un único perfil es bueno para afinar un banco de pruebas, pero puede olvidar algún detalle importante en una aplicación multipropósito. Peor todavía, si usted lo optimiza para un caso y lo mutila para otro, puede producir más daño que beneficio.

Perfilar, como hemos visto en este capítulo, es bastante mecánico. Afinar requiere de introspección. Es justo advertirle que no siempre es una actividad provechosa. A veces pone todo su empeño en una modificación inteligente que, contrariamente a lo esperado, incrementa el tiempo de ejecución. ¡Ah! ¿Qué salió mal? Dependerá usted de sus herramientas de perfilado para contestarse esta pregunta.

Cronometraje y Perfilado - Ejercicios

Exercise:

Problem:

Perfile el siguiente programa usando gprof. ¿Hay alguna forma de decir cuánto del tiempo gastado en la rutina c se debe a llamadas recursivas?

```
main()
{
    int i, n=10;
    for (i=0; i<1000; i++) {
        c(n);
        a(n);
    }
}
c(n)
int n;
{
    if (n > 0) {
        a(n-1);
        c(n-1);
    }
}
a(n)
int n;
{
    c(n);
}
```

Exercise:

Problem:

Perfile un código ingenieril (que haga uso intensivo de números de punto flotante) con la optimización completa activa y no activa. ¿Cómo cambia el perfil? ¿Puede explicar por qué?

Exercise:

Problem:

Escriba un programa para determinar la sobrecarga que representa invocar a getrusage y etime. Aparte del tiempo de procesador que consumen, ¿cómo altera el rendimiento de la aplicación, el hecho de invocar demasiado a menudo una llamada al sistema para cronometraje?

Eliminando el Desorden - Introducción

Hemos visto el código desde el punto de vista del compilador, y cómo perfilar código para encontrar las zonas problemáticas. Esta información es buena, pero si aún no está satisfecho con el rendimiento de cierto código, seguro piensa qué más hacer con él. Una posibilidad es que su código sea demasiado obtuso para que el compilador pueda optimizarlo apropiadamente: código excesivo, excesiva modularización o incluso "mejoras" debidas a procesos de optimización previos pueden desordenar su código y confundir a los compiladores. Se considera desorden a todo aquello que contribuye al tiempo de ejecución sin contribuir a la respuesta. Viene en dos formas:

Cosas que contribuyen a la sobrecarga
Llamados a subrutinas, referencias indirectas a memoria, pruebas adentro de bucles, pruebas excesivas, conversiones de tipos, variables preservadas innecesariamente

Cosas que restringen la flexibilidad del compilador
Llamados a subrutinas, referencias indirectas a memoria, pruebas adentro de bucles, apuntadores ambiguos

No se trata de un error que algunos elementos aparezcan repetidos en ambas listas. Tanto la invocación de subrutinas como el uso de sentencias condicionales adentro de bucles pueden perjudicarlo, al tomar mucho tiempo y generar *barreras* — lugares del programa donde las instrucciones que aparecen antes no pueden entremezclarse de forma segura con las instrucciones que aparecen después, al menos no sin un gran cuidado. El objetivo de este capítulo es mostrarle cómo eliminar el desorden, de forma que pueda reestructurar todo aquello que está mal y lograr la ejecución más rápida posible. Hemos eliminado unos cuantos tópicos que debieran estar aquí, especialmente aquellos relativos a referencias a memoria, y los colocamos en capítulos posteriores donde se tratan como temas separados.

Antes de comenzar, queremos recordarle: conforme busca usted formas de mejorar lo que tiene, mantenga ojos y mente abiertos a la posibilidad de que pueda existir una manera fundamentalmente mejor de hacer algo -una técnica de ordenación más eficiente, un generador de números aleatorios,

etc. Un algoritmo diferente puede aportarle mayor incremento de velocidad que afinar el existente. Analizar tales algoritmos está más allá del alcance de este libro, pero lo que estamos discutiendo aquí debe ayudarle a reconocer el "buen" código, o ayudarle a codificar un nuevo algoritmo que le de el mejor rendimiento.

Eliminando el Desorden - Llamado a Subrutinas

Una empresa típica está llena de aterrorizantes ejemplos de sobrecarga. Digamos que un departamento ha preparado una pila de papelería, que debe completar otro departamento. ¿Qué debe usted de hacer para transferir tal trabajo? Primero, debe asegurarse que la parte que le tocó hacer esté completa; no puede pedirles que lo acepten si los materiales que requieren no están listos. Después, necesitará empaquetar los materiales - datos, formas, números de cuenta, etc. Y finalmente viene la transferencia oficial. Una vez que recibió lo que usted les envió, el otro departamento debe desempacarlo, hacer su trabajo, reempaquetarlo y enviarlo de vuelta.

Se malgasta una gran cantidad de tiempo moviendo trabajos entre departamentos. Por supuesto, si la sobrecarga es mínima comparada con la cantidad de trabajo útil que se hace, no hay gran problema. Pero pudiera ser más eficiente llevar a cabo los trabajos pequeños adentro del mismo departamento. Lo mismo puede afirmarse de la invocación de subrutinas y funciones. Si sólo entra y sale de los módulos muy de vez en cuando, la sobrecarga de guardar los valores en los registros y preparar la lista de argumentos no resultará significativa. Sin embargo, si está invocando repetidamente unas pocas subrutinas pequeñas, la sobrecarga puede mantenerlas siempre encabezando la lista del perfil. Puede que fuera mejor si el trabajo permaneciera donde está, en la subrutina emisora de la llamada.

Adicionalmente, los llamados a subrutinas inhiben la flexibilidad del compilador. Si se presenta la oportunidad, usted quiere que su compilador tenga la flexibilidad suficiente para entremezclar instrucciones que no son dependientes las unas de las otras. Y tales instrucciones se encuentran en ambas caras de un llamado a subrutina, en el invocador y en el invocado. Pero la oportunidad se pierde cuando el compilador no puede emparejarlas dentro de subrutinas y funciones. Y así, instrucciones que perfectamente pudieran solaparse, permanecen en sus respectivos lados de la barrera artificial.

Puede ser de ayuda que ilustremos el reto que representan las fronteras de las subrutinas mediante un ejemplo exagerado. Los siguientes bucles se ejecutan muy bien en una amplia variedad de procesadores:

```
DO I=1,N
  A(I) = A(I) + B(I) * C
ENDDO
```

El código de abajo realiza los mismos cálculos, pero observe qué hemos hecho:

```
DO I=1,N
  CALL MADD (A(I), B(I), C)
ENDDO
SUBROUTINE MADD (A,B,C)
A = A + B * C
RETURN
END
```

Cada iteración invoca a una subrutina para realizar una pequeña cantidad de trabajo, que antes estaba ubicado adentro del ciclo. Este es un ejemplo particularmente doloroso, porque involucra cálculos de punto flotante. La pérdida de paralelismo resultante, junto con la sobrecarga debida al llamado al procedimiento, puede producir código que se ejecuta 100 veces más lento. Recuerde, tales operaciones se colocan en líneas de espera, y toma cierta cantidad de tiempo "de recuperación" antes de que el rendimiento alcance siquiera una operación por ciclo de reloj. Si son pocas las operaciones de punto flotante que se realizan entre llamados a subrutinas, el tiempo gastado en llenar y vaciar las líneas de espera se volverá muy importante.

La invocación de subrutinas y funciones complica la habilidad que tiene el compilador de manejar eficientemente variables `COMMON` y `external`, retrasando su almacenamiento en memoria hasta el último momento posible. El compilador usa registros para almacenar los valores "vivos" de muchas variables. Cuando realiza una llamada, el compilador no puede

saber cuáles de las variables cambiarán en la subrutina que está declarada como `external` o `COMMON`, y por ello se ve forzado a almacenar en memoria cualquier variable `external` o `COMMON` que haya sido modificada, de modo que la subrutina invocada pueda encontrarla. De igual modo, tras el retorno de la llamada, las mismas variables tienen que cargarse nuevamente en los registros, porque el compilador no puede confiar en las copias antiguas que residen en los registros. La penalización de guardar y recuperar variables puede ser substancial, especialmente si son muchas. También puede resultar riesgoso si las variables que debieran ser locales fueron especificadas como `external` o `COMMON`, como sucede en el siguiente código:

```
COMMON /USELESS/ K
DO K=1,1000
  IF (K .EQ. 1) CALL AUX
ENDDO
```

En este ejemplo, K se ha declarado como una variable `COMMON`. Sólo se usa como contador del ciclo do-loop, así que realmente no hay razón para que sea otra cosa que local. Sin embargo, dado que está en un bloque `COMMON`, la llamada a `AUX` obliga al compilador a almacenar y recargar `K` cada iteración. Ello se debe a que se desconocen los efectos laterales que la llamada puede producir.

Hasta aquí, pareciera como si estuviéramos abonando el camino para ¡programas principales enormes, sin subrutinas o funciones! En absoluto. La modularidad es importante para mantener compacto y comprensible el código fuente. Y francamente, la necesidad de modularidad y facilidad de mantenimiento siempre es más importante que la necesidad de *pequeñas* mejoras de rendimiento. Sin embargo, hay un par de enfoques que permiten racionalizar los llamados a subrutinas, y que no requieren que usted abandone las técnicas de codificación modular: se trata de las macros y los procedimientos en línea.

Recuerde, si la función o subrutina hace una cantidad razonable de trabajo, no importará demasiado la sobrecarga debida a la invocación del procedimiento. Sin embargo, si una rutina pequeña aparece como un nodo hijo en una de las secciones más atareadas del grafo de llamados, puede que deba pensar en insertarla en los lugares apropiados del programa.

Macros

Las *Macros* (o macroinstrucciones) son pequeños procedimientos que se substituyen en línea a tiempo de compilación. Al contrario que las subrutinas o funciones, que se incluyen una vez durante el proceso de enlazado, las macros se replican en cada lugar que se usan. Cuando el compilador lleva a cabo su primera pasada a lo largo de su programa, busca patrones que coincidan con las definiciones previas de macros, y las expande en línea. De hecho, en etapas posteriores, el compilador ve una macro expandida como si fuera código fuente creado por usted.

Las macros forman parte tanto de C como de FORTRAN (aunque la noción equivalente a una macro en FORTRAN, la *función sentencia*, ha sido vilipendiada por la comunidad FORTRAN y no sobrevivirá mucho tiempo más).[footnote] En los programas en lenguaje C, las macros se crean mediante una directiva `#define`, como se demuestra aquí:
La función sentencia se ha eliminado de FORTRAN 90.

```
#define promedio(x,y) ((x+y)/2)
main ()
{
   float q = 100, p = 50;
   float a;
   a = promedio(p,q);
   printf ("%f\n",a);
}
```

El primer paso de compilación de un programa en C consiste en pasar por el preprocesador de C, *cpp*. Ello ocurre automáticamente cuando usted invoca al compilador. *cpp* expande las sentencias `#define` en línea, reemplazando el patrón coincidente por la definición de la macro. En el programa de arriba, la sentencia:

```
a = promedio(p,q);
```

es reemplazada por:

```
a = ((p+q)/2);
```

Debe usted ser cuidadoso al definir la macro, porque literalmente reemplaza al patrón que *cpp* encuentra. Por ejemplo, si la definición de la macro decía:

```
#define multiplicar(a,b) (a*b)
```

y usted lo invocó como:

```
c = multiplicar(x+t,y+v);
```

la expansión resultante será `x+t*y+v` — que probablemente no es lo que usted pretendía.

Si es usted un programador en C, puede que esté usando macros sin siquiera percatarse. Muchos archivos de cabecera en C (.*h*) contienen definiciones

de macros. De hecho, algunas funciones "estándar" de biblioteca en C en realidad son macros contenidas en tales archivos. Por ejemplo, la función *getchar* puede enlazarse cuando construye su programa. Si tiene una sentencia:

```
#include <stdio.h>
```

en su archivo, *getchar* se reemplazará con una definición de macro a tiempo de compilación, reemplazando la función de biblioteca de C.

También puede usted hacer que las macros de *cpp* trabajen para los programas en FORTRAN.[footnote] Por ejemplo, una versión FORTRAN del programa en C anterior pudiera verse así:
Algunos programadores prefieren usar para FORTRAN el preprocesador estándar de UNIX, *m4* .

```
#define PROMEDIO(X,Y) ((X+Y)/2)
C
      PROGRAM MAIN
      REAL A,P,Q
      DATA P,Q /50.,100./
      A = PROMEDIO(P,Q)
      WRITE (*,*) A
      END
```

Sin algo de preparación, la sentencia `#define` será rechazada por el compilador de FORTRAN. El programa debe preprocesarse previamente mediante *cpp* para reemplazar el uso de `PROMEDIO` con su definición de macro. Ello convierte a la compilación en un procedimiento de dos pasos, pero que no debe ser una carga mayor, especialmente si está construyendo sus programas bajo el control del programa de utilidad *make*. También le sugerimos almacenar los programas en FORTRAN que contienen directivas

para *cpp* mediante la nomenclatura *filename.F* para distinguirlos de los programas FORTRAN sin adornos. Sólo asegúrese de hacer los cambios sólo a los archivos *.F*, y no a la salida de *cpp*. Así es como debe preprocesar los archivos *.F* de FORTRAN a mano:

```
% /lib/cpp -P < promedio.F > promedio.f
% f77 promedio.f -c
```

El compilador de FORTRAN nunca ve el código original. En vez de ello, la definición de la macro se sustituye en línea, tal y como si la hubiera tecleado usted mismo:

```
C
      PROGRAM MAIN
      REAL A,P,Q
      DATA P,Q /50.,100./ A = ((P+Q)/2)
      WRITE (*,*) A
      END
```

Por cierto, algunos compiladores de FORTRAN también reconocen la extensión *.F*, haciendo innecesario el proceso en dos pasos. Si el compilador ve la extensión *.F* invoca automáticamente a *cpp*, y desecha el archivo *.f* intermedio. Trate de compilar un archivo *.F* en su computadora para ver si funciona.

También, tome conciencia de que la expansión de macros puede provocar que las líneas de código fuente excedan la columna 72, lo cuál hará que su compilador de FORTRAN reclame (o peor, que le pase desapercibido). Algunos compiladores soportan líneas de entrada más largas de 72 caracteres. En los compiladores de Sun la opción **–e** permite extender las líneas de entrada hasta 132 caracteres de longitud.

Procedimientos En Línea

Las definiciones de macros tienden a ser muy cortas, usualmente de sólo una sentencia de longitud. A veces tiene usted porciones de código ligeramente más largas (pero no demasiado largas), que también pueden obtener beneficio si se copian en línea, en vez de invocarse como subrutina o función. Nuevamente, la razón de hacerlo es eliminar la sobrecarga debida a la invocación de procedimientos, y exponer el paralelismo. Si su compilador es capaz de definir subrutinas y funciones *inline* (en línea) al interior de los módulos que las invocan, entonces dispone de una forma muy natural y portátil de escribir código modular sin sufrir del costo debido a la invocación de la subrutina.

Dependiendo del vendedor, puede preguntarle al compilador si soporta la declaración de procedimientos en línea mediante:

- Especificar desde la línea de comandos aquellas rutinas que deben insertarse en línea
- Poner directivas para la declaración en línea dentro del código fuente
- Permitir que el compilador decida automáticamente qué poner en línea

Las directivas y las opciones de la línea de comandos del compilador no están estandarizadas, así que deberá revisar la documentación de su compilador. Desafortunadamente, puede que nuevamente descubra que no hay tal característica (“nuevamente,” siempre nuevamente), o que es una característica extra que por la que deberá pagar. La tercera forma de declaración en línea de la lista, la automática, sólo la tienen disponible unos pocos vendedores. La declaración en línea automática depende de que el compilador sea sofisticado y pueda ver las definiciones de varios módulos a la vez.

Unas últimas palabras de precaución respecto a la colocación de procedimientos en línea. Es fácil exagerar en su uso. Si todo termina enquistado en el cuerpo o en sus padres, el ejecutable resultante puede ser tan grande que repetidamente desborde la cache de instrucciones y se convierta en una pérdida de rendimiento neto. Nuestro consejo es que use la información de invocador/invocado que le proporcionan los perfiladores para tomar decisiones inteligentes acerca de la ubicación en línea, en vez de

tratar de aplicarlo a cada subrutina que genere. De nuevo, los mejores candidatos generalmente son aquellas rutinas pequeñas que se invocan muy a menudo.

Eliminando el Desorden - Bifurcaciones

A veces a las personas les lleva una semana tomar una decisión, así que no deberíamos quejarnos si una computadora se toma unas pocas decenas de nanosegundos para hacerlo. Sin embargo, si aparece una sentencia selectiva en alguna sección altamente visitada del código, puede que usted acabe cansado de los retrasos que produce. Hay dos enfoques básicos para reducir el impacto de estas bifurcaciones:

- Racionalizarla.
- Enviarla a los suburbios computacionales. Particularmente, mandarla afuera de los bucles.

[link] mostramos algunas formas fáciles de reorganizar condicionales, de forma que se ejecuten más rápidamente.

Eliminando el Desorden - Bifurcaciones con Ciclos

Los códigos numéricos usualmente gastan la mayoría de su tiempo de ejecución en ciclos, así que usted no quiere nada adentro de tales ciclos que no deba estar ahí, especialmente una sentencia selectiva. Y no sólo se trata de que las sentencias selectivas entorpezcan el trabajo con instrucciones extras; también pueden forzar un orden estricto sobre las iteraciones de un ciclo. Por supuesto, no siempre pueden evitarse tales condicionales. A veces, sin embargo, la gente las pone adentro de los ciclos para procesar eventos que pudieran manejarse por separado, o incluso ignorarse.

Para hacerle retroceder unos pocos años, el siguiente código muestra un ciclo con una prueba para un valor cercano a cero:

```
PARAMETER (SMALL = 1.E-20)
DO I=1,N
  IF (ABS(A(I)) .GE. SMALL) THEN
    B(I) = B(I) + A(I) * C
  ENDIF
ENDDO
```

La idea era que si el factor, `A(I)`, era razonablemente pequeño, no se justificaba realizar la matemática en el centro del ciclo. Dado que las operaciones de punto flotante no se colocaban en filas de espera en muchas máquinas, una comparación y un salto eran más económicos; la prueba ahorraba tiempo. En un procesador CISC más antiguo o en los primeros RISC, una comparación y un salto probablemente aún ahorren tiempo. Pero en otras arquitecturas, cuesta mucho menos realizar la matemática y evitar la prueba. Eliminar la bifurcación elimina una dependencia de control y permite que el compilador ponga en fila de espera más operaciones aritméticas. Por supuesto, la respuesta puede cambiar ligeramente si se elimina la prueba. Luego se convierte en una cuestión de cuán significativa es la diferencia. He aquí otro ejemplo donde no se requiere de una bifurcación. El bucle encuentra el valor absoluto de cada elemento en un arreglo:

```
DO I=1,N
  IF (A(I) .LT. 0.) A(I) = -A(I)
ENDDO
```

Pero, ¿por qué realizar la prueba? En muchas máquinas, es más rápido aplicar la operación `abs()` a cada elemento del arreglo.

Debemos, empero, hacerle una advertencia: si está programando en C, el valor absoluto `fabs()`, es un llamado a subrutina. En este caso particular, le conviene más mantener el condicional adentro del ciclo.[footnote]
La representación de la máquina de un número de punto flotante comienza con un bit de signo. Si el bit es cero, el número es positivo. Si es 1, el número es negativo. La función de valor absoluto más rápida es aquella que simplemente aplique un "Y" al bit de signo. Véanse los macros en */usr/include/macros.h* y */usr/include/math.h*.

Aún en aquellos casos en que no pueda deshacerse del condicional, todavía quedan cosas que puede hacer para minimizar la pérdida de rendimiento. Primero, debemos aprender a reconocer cuáles condicionales dentro de los bucles pueden reestructurase y cuáles no. Las sentencias selectivas adentro de bucles caen en varias categorías:

- Condicionales invariantes en el ciclo
- Condicionales dependientes del índice del ciclo
- Condicionales independientes del ciclo
- Condicionales dependientes del ciclo
- Reducciones
- Condicionales que transfieren el control

Revisémoslas una por una.

Condicionales Invariantes en el Ciclo

El siguiente ciclo contiene una prueba *invariante*:

```
DO I=1,K
  IF (N .EQ. 0) THEN
    A(I) = A(I) + B(I) * C
  ELSE
    A(I) = 0.
  ENDIF
ENDDO
```

“Invariante” significa que el resultado siempre es el mismo. Sin importar lo que suceda con las variables `A`, `B`, `C`, e `I`, el valor de `N` no cambia, ni tampoco lo hará el resultado del resto.

Usted puede reestructurar el ciclo, poniendo la prueba fuera y replicando el cuerpo del ciclo dos veces - una para el caso de que la prueba sea verdadera, y la otra para el resultado falso, como en el ejemplo siguiente:

```
IF (N .EQ. 0) THEN
  DO I=1,K
    A(I) = A(I) + B(I) * C
  ENDDO
ELSE
  DO I=1,K
    A(I) = 0
  ENDDO
ENDIF
```

El efecto en el tiempo de ejecución es dramático. No sólo hemos eliminado `K-1` copias de la prueba, también hemos asegurado que los cálculos en el centro del ciclo no tengan dependencias de control sobre la sentencia selectiva, y por tanto es mucho más fácil que el compilador lo ponga en una fila de espera.

Recordamos haber ayudado a alguien a optimizar un programa con ciclos que contenían condicionales similares. Comprobaban si debía imprimirse la salida del depurador adentro de cada iteración, o bien producir un ciclo altamente optimizable. No podemos culpar a la persona por no observar cuánto reducía esto la velocidad del programa. El rendimiento no era importante en ese momento. El programador sólo trataba de lograr que el código produjera respuestas correctas. Pero después, cuando comenzó a importar el rendimiento, limpiando condicionales invariantes fuimos capaces de acelerar el programa en un factor de 100.

Condicionales Dependientes del Indice del Ciclo

Para los condicionales *dependientes del índice del ciclo*, la prueba es verdadera sólo para ciertos rangos de las variables usadas como índice del ciclo. Y no siempre es cierto o falso, como en otros condicionales que ya revisamos, pero cambia de acuerdo a un patrón apreciable, uno que puede usarse a nuestro favor. El siguiente ciclo tiene dos variables índice, `I` y `J`.

```
DO I=1,N
  DO J=1,N
    IF (J .LT. I)
      A(J,I) = A(J,I) + B(J,I) * C
    ELSE
      A(J,I) = 0.0
    ENDIF
  ENDDO
ENDDO
```

Observe cómo la sentencia selectiva divide las iteraciones en dos conjuntos distintos: aquéllas en las cuáles es verdadero y aquéllas en las que es falso. Puede usted tomar ventaja de que tal cosa sea predecible, para reestructurar el bucle en varios bucles - cada uno personalizado para una partición distinta:

```
      DO I=1,N
        DO J=1,I-1
            A(J,I) = A(J,I) + B(J,I) * C
        ENDDO
        DO J=I,N
            A(J,I) = 0.0
        ENDDO
      ENDDO
```

La nueva versión resultará más rápida casi siempre, con la posible excepción del caso en que `N` tenga un valor pequeño, como 3, en cuyo caso hemos creado un mayor desorden. Pero aun en tal caso, el ciclo probablemente tendrá un impacto pequeño en el tiempo total de ejecución, del que hubiera tenido si se dejaba tal como está codificado.

Condicionales Independientes del Ciclo

Sería bueno que pudiera optimizar cada ciclo, particionándolo. Pero muy a menudo, el condicional no depende directamente del valor de las variables índice. Aunque una variable índice esté involucrada en el direccionamiento de un arreglo, no crea por adelantado un patrón reconocible - por lo menos no uno que pueda usted observar cuando está escribiendo el programa. He aquí uno de tales ciclos:

```
      DO I=1,N
        DO J=1,N
          IF (B(J,I) .GT. 1.0) A(J,I) = A(J,I) +
B(J,I) * C
        ENDDO
      ENDDO
```

NO hay mucho que pueda hacer con este tipo de condicional. Pero dado que cada iteración es independiente, puede desenrollar el ciclo, o bien realizarlo en paralelo.

Condicionales Dependientes del Ciclo

Cuando el condicional se basa en un valor que cambia con cada iteración del ciclo, al compilador no le queda más opción que ejecutar el código exactamente tal y como se escribió. Por ejemplo, el siguiente ciclo tiene una sentencia selectiva con una recursividad escalar interna:

```
DO I=1,N
  IF (X .LT. A(I)) X = X + B(I)*2.
ENDDO
```

No puede usted saber en qué forma actuará la bifurcación en la siguiente iteración hasta que no haya ejecutado la actual. Para reconocer la dependencia, trate de desenrollar ligeramente el ciclo manualmente. Si no puede comenzar la segunda prueba hasta haber finalizado la primera, tiene un condicional *dependiente del ciclo*. Puede que quiera revisar este tipo de bucles en busca de formas de eliminar el valor que cambia de iteración a iteración.

Reducciones

Mantenga la vista atenta en busca de ciclos en los cuales la sentencia selectiva está aplicando una función max o min a un arreglo. Se trata de una *reducción*, así llamada porque reduce todo un arreglo a un resultado escalar (por cierto, el ejemplo previo también fue una reducción). De nuevo, estamos un poco por delante de nosotros mismos, pero dado que estamos hablando acerca de sentencias selectivas adentro de bucles, quiero introducir un truco para reestructurar las reducciones del tipo `max` y `min` para exponer un mayor paralelismo. El siguiente ciclo busca el valor máximo, `z`, en el arreglo `a`, recorriendo todos los elementos uno a la vez:

```
for (i=0; i<n; i++)
    z = a[i] > z ? a[i] : z;
```

Tal como está escrito es recursivo, como el bucle de la sección previa. Requiere usted el resultado de una iteración dada antes de que pueda proceder con la siguiente. Sin embargo, y dado que estamos buscando el mayor elemento en todo el arreglo, y dado que será el mismo elemento (esencialmente) sin importar dónde lo hallemos, podemos reestructurar el ciclo para comprobar varios elementos a la vez (asumiendo que n es divisible de forma entera entre 2, y que no incluya el ciclo precondicionante):

```
z0 = 0.;
z1 = 0.;
for (i=0; i< n-1; i+=2) {
  z0 = z0 < a[i] ? a[i] : z0;
  z1 = z1 < a[i+1] ? a[i+1] : z1;
}
z = z0 < z1 ? z1 : z0;
```

¿Observa cómo calcula el nuevo ciclo dos valores máximos en cada iteración? Estos máximos se comparan luego uno con otro, y el ganador se convierte en el nuevo valor *max* oficial. Es análogo a la ronda de semifinales en un torneo de ping-pong. El ciclo anterior consistía en dos jugadores compitiendo a la vez mientras que el resto se sentaba a su alrededor; en cambio, el nuevo ciclo ejecuta varios encuentros simultáneos. En general esta optimización en particular no es adecuada para codificarse a mano. En los procesadores paralelos, el compilador realiza la reducción a su propio modo. Si usted lo codificó de forma similar a este ejemplo, puede que esté limitando inadvertidamente la flexibilidad del compilador en un sistema paralelo.

Condicionales que Transfieren el Control

Detengámonos por un segundo. ¿Ha notado cierta similaridad en todos los bucles hasta ahora? Sólo hemos buscado un tipo de condicional, la *asignación condicional*, esto es, la reasignación de una variable basados en el resultado de la prueba. Por supuesto, no todo condicional termina en una asignación. Existen sentencias que transfieren el control de flujo, tales como llamados a subrutinas o sentencias `goto`. En el siguiente ejemplo, el programador está comprobando cuidadosamente antes de dividir entre cero:

Pero esta prueba tiene un impacto extremadamente negativo en el rendimiento, porque fuerza la ejecución de las iteraciones precisamente en el orden en que se escriben:

```
DO I=1,N
  DO J=1,N
    IF (B(J,I) .EQ. 0 ) THEN
      PRINT *,I,J
      STOP
    ENDIF
    A(J,I) = A(J,I) / B(J,I)
  ENDDO
ENDDO
```

Evitar este tipo de pruebas es una de las razones por las cuales los diseñadores del estándar de punto flotante del IEEE agregaron las trampas (traps) en operaciones tales como dividir entre cero. El uso de trampas permite al programador dentro de una sección de código de rendimiento crítico, lograr un rendimiento máximo sin necesidad de detectar cuándo ocurre un error.

El desorden puede asumir muchas formas. Considere que en las secciones previas hemos batallado con grandes piezas de basura como las que puede usted encontrar en el armario de su sala: una tabla de planchar, palos de hockey y palos de billar. Ahora nos dedicaremos a cosas más pequeñas: una goma de borrar, una pelota de tenis y el sombrero que nadie usa. Aquí queremos mencionar unos cuantos. De entrada pedimos disculpas por cambiar los temas tanto, ¡pero así es la naturaleza de limpiar un armario!

Conversiones de Tipos de Datos

Las sentencias que incluyen conversiones de tipos a tiempo de ejecución producen algo de penalización del rendimiento cada vez que se ejecutan. Si la sentencia se localiza en una porción muy activa del programa, la penalización total puede resultar significativa.

Las personas tienen sus razones para escribir aplicaciones que mezclan tipos. A menudo es cuestión de ahorrar espacio de memoria, ancho de banda de memoria, o tiempo. En el pasado, por ejemplo, los cálculos de doble precisión tomaban el doble que sus contrapartes de precisión sencilla, así que si alguno de tales cálculos podía arreglarse para hacerse en precisión sencilla, beneficiaba al rendimiento.[footnote] Pero cualquier tiempo ahorrado realizando parte de los cálculos en precisión sencilla y la otra parte en doble precisión, debe medirse contra la sobrecarga adicional causada por las conversiones de tipos a tiempo de ejecución. En el siguiente código, la suma de `A(I)` con `B(I)` tiene *tipos mezclados*:
Actualmente, los cálculos en precisión sencilla pueden tardar *más tiempo* que aquellos en doble precisión de registro a registro.

```
INTEGER NUMEL, I
PARAMETER (NUMEL = 1000)
REAL*8 A(NUMEL)
REAL*4 B(NUMEL)
DO I=1,NUMEL
```

```
  A(I) = A(I) + B(I)
ENDDO
```

En cada iteración, `B(I)` debe convertirse a doble precisión antes de que pueda ocurrir la suma. Usted no ve la conversión en el código fuente, pero está ahí, y consume tiempo.

Advertencia para los programadores de C: en el libro de C de Kernighan y Ritchie (K&R) C, todos los cálculos de punto flotante que aparecen en los programas son en doble precisión - incluso si todas las variables involucradas están declaradas como *float*. Es posible que usted escriba una aplicación K&R completa en una precisión, si bien sufrirá la penalización derivada de muchas conversiones de tipos.

Otro error relacionado con los tipos de datos, es usar operaciones de caracteres en las pruebas de las sentencias IF. En muchos sistemas, las operaciones sobre caracteres tienen un rendimiento más pobre que sus equivalentes enteras, dado que se realizan mediante llamados a procedimientos. Por lo mismo, puede que los optimizadores no revisen el código que usa variables de tipo caracter como buenos candidatos para optimización. Por ejemplo, el siguiente código:

```
DO I=1,10000
  IF ( CHVAR(I) .EQ. 'Y' ) THEN
    A(I) = A(I) + B(I)*C
  ENDIF
ENDDO
```

puede escribirse mejor usando una variable entera para indicar si debe o no realizarse el cálculo:

```
DO I=1,10000
```

```
  IF ( IFLAG(I) .EQ. 1 ) THEN
    A(I) = A(I) + B(I)*C
  ENDIF
ENDDO
```

Otra forma de escribir el código, asumiendo que la variable IFLAG fue 0 o 1, sería la siguiente:

```
DO I=1,10000
  A(I) = A(I) + B(I)*C*IFLAG(I)
ENDDO
```

El último enfoque puede de hecho provocar mayor lentitud en algunos sistemas de cómputo, respecto al primero, sobre todo si éste usa la variable entera en la sentencia IF.

Haciendo su Propia Eliminación de Subexpresiones Comunes

Hasta ahora le hemos dado a su compilador el beneficio de la duda. La *eliminación de subexpresiones comunes* — la habilidad del compilador de reconocer patrones repetidos en el código, y reemplazar todos excepto uno por una variable temporal- probablemente trabaje en su máquina con expresiones simples. En la siguientes líneas de código, la mayoría de los compiladores reconocerán `a+b` como una subexpresión común:

```
c = a + b + d
e = q + a + b
```

que se transforma en:

```
temp = a + b
c = temp + d
e = q + temp
```

La sustitución de `a+b` elimina algo de la aritmética. Si la expresión se reutiliza muchas veces, los ahorros pueden resultar significativos. Sin embargo, es limitada la habilidad de un compilador para reconocer subexpresiones comunes, especialmente cuando tienen componentes múltiples o éstos aparecen permutados. Un compilador puede no ser capaz de reconocer que `a+b+c` y `c+b+a` son equivalentes.[footnote] En las partes más importantes del programa, debe usted considerar la eliminación a mano de las subexpresiones comunes en las expresiones más complicadas. Ello garantiza que se realizará. En cierto modo compromete la belleza, pero hay algunas situaciones donde es mejor.
Y por causa del sobreflujo y los errores de redondeo de punto flotante, en algunas situaciones pueden no ser equivalentes.

He aquí otro ejemplo en el cuál la función seno se invoca dos veces con el mismo argumento:

```
x = r*sin(a)*cos(b);
y = r*sin(a)*sin(b);
z = r*cos(a);
```

y que se convierte en:

```
temp = r*sin(a);
x = temp*cos(b);
y = temp*sin(b);
```

```
z = r*cos(a);
```

Hemos reemplazado uno de los llamados por una variable temporal. Estamos de acuerdo en que los ahorros de eliminar la invocación de una función trascendente de entre cinco no nos hará ganar el Premio Nobel, pero nos hace prestar atención a un punto importante: los compiladores típicamente no realizan la eliminación de subexpresiones comunes en subrutinas o llamados a funciones. El compilador no puede estar seguro que el llamado a la rutina no cambie el estado del argumento o de alguna otra variable que no puede ver.

La única ocasión en que un compilador puede eliminar subexpresiones comunes que contengan llamados a funciones, es cuando son intrínsecas, como en FORTRAN. Y puede hacerse porque el compilador puede asumir ciertas cosas acerca de los efectos laterales. Usted, por otra parte, puede ver adentro de las subrutinas, lo cuál significa que está mejor calificado que el compilador para agrupar subexpresiones comunes que involucren subrutinas o funciones.

Llevando a Cabo sus Propias Reubicaciones de Código

Donde mejor resulta llevar a cabo estas optimizaciones es adentro de los ciclos, porque es donde se concentra toda la actividad de un programa. Una de las mejores formas de recortar el tiempo de ejecución es mover las instrucciones innecesarias o repetidas (invariantes) fuera del flujo principal del código, a los suburbios. En el caso de los bucles, se le denomina *emerger* instrucciones cuando se desplazan hacia la parte superior y *sumergir* instrucciones cuando se colocan en la parte inferior. He aquí un ejemplo:

```
DO I=1,N
  A(I) = A(I) / SQRT(X*X + Y*Y)
ENDDO
```

que se convierte en:

```
TEMP = 1 / SQRT(X*X + Y*Y)
DO I=1,N
  A(I) = A(I) * TEMP
ENDDO
```

Hicimos emerger una operación cara e invariante hacia afuera del ciclo, y asignamos el resultado a una variable temporal. Observe también que hicimos una simplificación algebraica cuando intercambiamos una división por la multiplicación por un inverso. La multiplicación se ejecutará mucho más rápido. Puede ser que su compilador sea lo suficientemente listo para hacer tales transformaciones por sí mismo, asumiendo que usted le haya instruido de que son transformaciones legales; pero sin rastrear el código ensamblador generado, no puede usted estar seguro. Por supuesto, si usted reacomoda el código a mano y el tiempo de ejecución del ciclo súbitamente disminuye, sabrá que el compilador ha sido un lastre hasta entonces.

En ocasiones querrá usted sumergir una operación tras el ciclo. Usualmente, se trata de algún cálculo realizado cada iteración, pero cuyo resultado sólo se requiere al final. Para ilustrarlo, he aquí una clase de ciclo distinto de los que hemos visto hasta ahora, uno que busca el caracter final en una cadena:

```
while (*p != ' ')
  c = *p++;
```

se convierte en:

```
while (*p++ != ' ');
```

```
c = *(p-1);
```

La nueva versión del ciclo mueve la asignación de `c` más allá de la última iteración. Debemos admitir que esta transformación puede ser un logro para un compilador y que el ahorro no sea muy grande. Pero ilustra la idea de sumergir una operación muy bien.

Nuevamente, emerger y sumergir instrucciones para colocarlas afuera de ciclos es algo que su compilador puede que sea capaz de hacer. Pero a menudo usted puede reestructurar ligeramente los cálculos por sí mismo sólo con moverlos, y obtener un beneficio todavía mayor.

Manipulando los Elementos de un Arreglo en Bucles

Hay otra área donde debe usted confiar en que el compilador hará lo correcto. Cuando emplee repetidamente un elemento de un arreglo adentro de un bucle, querrá que se cargue una sola vez de la memoria. Tome el siguiente bucle como un ejemplo. Reutiliza `X(I)` dos veces:

```
DO I=1,N
  XOLD(I) = X(I)
  X(I)= X(I) + XINC(I)
ENDDO
```

En realidad, los pasos encargados de recuperar `X(I)` sólo son subexpresiones comunes adicionales: un cálculo de direcciones (posiblemente) y una operación de carga de memoria. Puede usted ver que la operación se repite al reescribir el ciclo ligeramente:

```
DO I=1,N
  TEMP= X(I)
```

```
  XOLD(I) = TEMP
  X(I)= TEMP + XINC(I)
ENDDO
```

Los compiladores de FORTRAN *deben* reconocer que se está usando dos veces el mismo elemento `X(I)` y que por tanto sólo se requiere cargarlo una vez, pero no siempre un compilador es tan inteligente. A veces tiene usted que crear una variable escalar temporal para almacenar el valor de un elemento del arreglo en el cuerpo de un bucle. Esto es particularmente cierto cuando hay llamados a subrutinas o funciones en el ciclo, o cuando alguna de las variables está declarada como `external` o `COMMON`. Asegúrese de emparejar los tipos entre las variables temporales y las otras variables, pues no quiere incurrir la sobrecarga derivada de la conversión de tipos sólo por estar "ayudando" al compilador. En el caso de los compiladores de C, el mismo tipo de expresiones indexadas son un reto incluso mayor. Considere este código:

```
doinc(int xold[],int x[],int xinc[],int n)
{
  for (i=0; i<n; i++) {
    xold[i] = x[i];
    x[i]= x[i] + xinc[i];
  }
}
```

A menos que el compilador pueda ver las definiciones de `x`, `xinc` y `xold`, debe asumir que son apuntadores señalando a la misma celda de almacenamiento, y repetir las operaciones de carga y almacenamiento. En este caso, introducir variables temporales para almacenar los valores de `x`, `xinc`, y `xold` es una optimización que el compilador no es libre de hacer.

Es interesante señalar que si bien usar variables escalares temporales en el ciclo es útil para las máquinas RISC y superescalares, no ayuda al código

que se ejecuta sobre hardware paralelo. Un compilador paralelo busca oportunidades de eliminar los escalares o, cuando menos, de reemplazarlos con vectores temporales. Si ejecuta su código sobre una máquina paralela de vez en cuando, debe ser cuidadoso antes de introducir variables escalares temporales en un ciclo. Una dudosa ganancia de rendimiento en una instancia puede convertirse en una pérdida de rendimiento real en otra.

Eliminando el Desorden - Notas de Cierre

En este capítulo, introdujimos técnicas de afinación que ayudan a eliminar el desorden de los programas - cualquier cosa que agregue tiempo de ejecución sin contribuir a hallar la respuesta. Hemos visto muchos ejemplos de técnicas de afinación -suficientes como para que usted se pregunte, "¿qué puede faltar?" Bueno, como veremos en los capítulos siguientes, todavía restan un par de formas en que podemos ayudar al compilador:

- Encontrando más paralelismo
- Empleando la memoria tan eficientemente como sea posible

A veces ello significará realizar cambios que no son bonitos. Sin embargo, a menudo sí resultan rápidos.

Eliminando el Desorden - Ejercicios

Exercise:

Problem:

¿Cómo simplificaría usted el siguiente ciclo condicional?
`DO I=1,N A(I) = A(I) * B IF (I .EQ. N/2) A(I) = 0. ENDDO`

Exercise:

Problem:

Cronometre el siguiente ciclo en su computadora, con y sin la condición. Ejecútelo con tres conjuntos de datos: uno con todos los valores `A(I)` menores que `SMALL`, uno con todos los valores `A(I)` mayores que `SMALL`, y un tercero dividido en partes iguales. ¿Cuándo es mejor dejar la prueba en el ciclo, si es el caso?
`PARAMETER (SMALL = 1.E-20) DO I=1,N IF (ABS(A(I)) .GE. SMALL) THEN B(I) = B(I) + A(I) * C ENDIF ENDDO`

Exercise:

Problem:

Escriba un programa sencillo que invoque a una subrutina sencilla en su ciclo interno. Cronometre la ejecución del programa. Luego dígale al compilador que compile la rutina en línea, y cronometre nuevamente. Finalmente, modifique el código para realizar las operaciones en el cuerpo del ciclo, y cronometre el código. ¿Cuál opción se ejecutó más rápido? Puede que deba revisar el código máquina generado para imaginar el porqué.

Optimización de Ciclos - Introducción

En prácticamente todas las aplicaciones de alto rendimiento, es en los bucles donde se gasta la mayoría del tiempo de ejecución. [link] examinamos formas en las que los desarrolladores de la aplicación introdujeron desorden en los bucles, posiblemente haciéndolos ejecutarse más lento de lo necesario. En este capítulo nos enfocaremos en técnicas para mejorar el rendimiento de tales ciclos "libres de desorden". A veces el compilador es lo suficientemente inteligente para generar las versiones más rápidas de los bucles, y otras veces nos veremos en la necesidad de reescribirlos manualmente para ayudarle.

Es importante recordar que las modificaciones que hagamos para mejorar el rendimiento del compilador, provocan otro desorden en el compilador. Cuando hace usted modificaciones en nombre del rendimiento, debe asegurarse que éstas realmente ayudan, probando el rendimiento con y sin ellas. También, cuando se mueva a otra arquitectura, necesita asegurarse que cualquier modificación hecha no perjudique al rendimiento. Por esta razón, debe elegir sabiamente las modificaciones que haga en pos del rendimiento. También debe dejar intacta la versión original (simple) del código, para probarla en nuevas arquitecturas. Si nota que el beneficio derivado de las modificaciones es pequeño, muy probablemente deba mantener el código en su versión más simple y clara.

Revisaremos varias técnicas de optimización de bucles diferentes, incluyendo:

- Desenrollado de bucles
- Optimización de bucles anidados
- Intercambio de bucles
- Optimización de las referencias a memoria
- Bloqueo
- Soluciones fuera del núcleo

Tal vez algún día sea posible que un compilador realice todas estas optimizaciones de bucles automáticamente. Es normal que el desenrollado de bucles se realice como parte de las optimizaciones de un compilador normal. Otras puede que deban incluirse mediante opciones explícitas del

compilador al invocarlo. Cuando contemple hacer cambios manualmente, observe cuidadosamente cuáles de tales optimizaciones hace el compilador automáticamente. Y ejecute algunas pruebas para determinar si las optimizaciones del compilador son tan buenas como aquellas realizadas a mano.

Optimización de Ciclos - Conteo de Operaciones

Antes de que comience a reescribir el cuerpo de un bucle, o reorganizar el orden de los bucles, debe tener alguna idea de qué hace dicho cuerpo en cada iteración. Se llama *conteo de operaciones* al proceso de medir un ciclo para comprender la mezcla de operaciones. Necesitará contar el número de cargas, almacenamiento, operaciones de punto flotante, enteras, y llamados a bibliotecas por cada iteración del bucle. A partir de este conteo, puede ver cuán bien se corresponde la mezcla de operaciones de un ciclo dado con las capacidades del procesador. Por supuesto, el conteo de operaciones no garantiza que el compilador generará una representación eficiente de un ciclo.[footnote] Pero ello generalmente le proporcionará una percepción suficiente como para dirigir sus esfuerzos de afinación.
Revise la salida en lenguaje ensamblador para estar seguro, la cuál puede estar introduciendo algo de desbordamiento. Para obtener un listado en lenguaje ensamblador en muchas máquinas, compile con la bandera **–S**. En las RS/6000, utilice la bandera **–qlist**.

Tenga en mente que una mezcla de instrucciones balanceada para una máquina puede no estarlo para otra. Los procesadores que hay hoy día en el mercado pueden generalmente ejecutar alguna combinación de una a cuatro instrucciones por ciclo de reloj. La aritmética de direcciones a menudo queda incrustada en las instrucciones que hacen referencia a memoria. Como el compilador puede reemplazar complicados cálculos de direcciones de bucles por expresiones simples (suponiendo que el patrón de direcciones es predecible), en ocasiones puede usted ignorar la aritmética de direcciones cuando hace el conteo de operaciones.[footnote]
El compilador reduce la complejidad de las expresiones de indexación del ciclo con una técnica llamada *simplificación de variables de inducción*. Véase [link].

Revisemos unos cuantos bucles, y veamos qué podemos aprender acerca de la mezcla de instrucciones:

```
DO I=1,N
  A(I,J,K) = A(I,J,K) + B(J,I,K)
```

```
ENDDO
```

Este bucle contiene una suma de punto flotante y tres referencias a memoria (dos cargas y un almacenamiento). Hay algunas expresiones complejas de indexación de arreglos, pero probablemente el compilador las simplifique y pueda ejecutarlas en el mismo ciclo que aquellas de memoria y punto flotante. Para cada iteración del ciclo, debemos incrementar la variable índice y probarla para determinar si éste se ha completado.

Una tasa 3:1 de referencias a memoria respecto a operaciones de punto flotante sugiere que podemos aspirar a por lo menos 1/3 del rendimiento pico de punto flotante al ejecutar el bucle, a menos que haya más de un camino a la memoria. Son malas noticias, pero buena información. La tasa nos dice que debiéramos considerar primero la optimización de las referencias a memoria.

El bucle de abajo contiene una suma de punto flotante y dos operaciones a memoria - una carga y un almacenamiento. El operando `B(J)` es invariante en el bucle, así que su valor sólo requiere de cargarse una vez, antes de la entrada al bucle:

```
DO I=1,N
  A(I) = A(I) + B(J)
ENDDO
```

Nuevamente, nuestro rendimiento de punto flotante está limitado, pero no tan severamente como en el ciclo anterior. La tasa de referencias a memoria respecto a operaciones de punto flotante es ahora de 2:1.

El siguiente ejemplo muestra un bucle con mejores perspectivas. Realiza multiplicaciones, a nivel de elementos, de números complejos y asigna los resultados de vuelta al primero de ellos. Hay seis operaciones a memoria (cuatro cargas y dos almacenamientos) y seis operaciones de punto flotante (dos sumas y cuatro multiplicaciones):

```
for (i=0; i<n; i++) {
    xr[i] = xr[i] * yr[i] - xi[i] * yi[i];
    xi[i] = xr[i] * yi[i] + xi[i] * yr[i];
}
```

Pareciera que este bucle está cercanamente balanceado para un procesador que pueda realizar el mismo número de operaciones de memoria y de punto flotante por ciclo. Sin embargo, puede que no sea así. Muchos procesadores realizan una multiplicación y una suma de punto flotante en una sola instrucción. Si el compilador es lo suficientemente bueno para reconocer que la suma-multiplicación es apropiada, este ciclo puede también verse limitado por las referencias a memoria; cada iteración se compilará en dos multiplicaciones y dos multiplicaciones-sumas.

De nuevo, el conteo de operaciones es una forma simple de estimar cuán bien se mapean los requerimientos de un ciclo en las capacidades de una máquina. En el caso de muchos bucles, a menudo encontrará usted que su rendimiento está dominado por las referencias a memoria, tal como vimos en los últimos tres ejemplos. Ello sugiere que la afinación de las referencias a memoria es muy importante.

Optimización de Ciclos - Desenrollado Básico de Ciclos

La forma más básica de optimización de bucles es desenrollarlos. Es algo tan básico que casi todos los compiladores actuales lo hacen automáticamente, si da la impresión de que repercutirá benéficamente. En nombre del desenrollado de bucles, se introdujo gran cantidad de desorden en los antiguos programas FORTRAN ahora cubiertos de polvo, que sólo sirve para confundir y despistar a los compiladores de hoy.

No estamos sugiriendo que deba usted desenrollar los bucles a mano. El propósito de esta sección es doble: primero, una vez que esté familiarizado con el desenrollado de bucles, podrá reconocer código que desenrolló el programador (no usted) tiempo atrás, y simplificarlo; y segundo, necesita entender los conceptos del desenrollado de bucles, de modo que cuando vea el código máquina generado, pueda reconocerlos.

El beneficio primario de desenrollar bucles, es realizar más cálculos por iteración. Al final de cada una, el valor índice debe incrementarse, probarse y bifurcar el control de regreso al inicio del ciclo, si todavía faltan iteraciones por procesar. Al desenrollar el bucle, hay menos "finales de bucle" en la ejecución de cada ciclo. Desenrollar también reduce significativamente el número total de bifurcaciones, y da al procesador más instrucciones entre ellas (i.e., incrementa el tamaño de los bloques básicos).

A modo de ilustración, considere el siguiente bucle. Es una sentencia simple envuelta en un ciclo:

```
DO I=1,N
  A(I) = A(I) + B(I) * C
ENDDO
```

Usted puede desenrollar el bucle, como haremos en el ejemplo siguiente, reescribiendo las mismas operaciones mediante menos iteraciones y con menos sobrecarga en el ciclo. Imagine cómo ayudará esto en cualquier computadora. Dado que los cálculos en una iteración no dependen de los

cálculos en otras, los primeros y los segundos pueden ejecutarse juntos. En un procesador superescalar, ciertas porciones de estas cuatro sentencias pueden ejecutarse en paralelo:

```
DO I=1,N,4
  A(I) = A(I) + B(I) * C
  A(I+1) = A(I+1) + B(I+1) * C
  A(I+2) = A(I+2) + B(I+2) * C
  A(I+3) = A(I+3) + B(I+3) * C
ENDDO
```

Sin embargo, este ciclo no es *exactamente* el mismo que el anterior, pues se desenrolló cuatro veces, pero ¿qué pasa si `N` no es divisible entre 4? Habrá una, dos o tres iteraciones sobrantes que no se ejecutarán. Para manejar tales iteraciones extras, agregamos otro pequeño bucle para absorberlas. El ciclo extra se llama un *bucle precondicionante*:

```
II = IMOD (N,4)
DO I=1,II
  A(I) = A(I) + B(I) * C
ENDDO

DO I=1+II,N,4
  A(I)   = A(I) + B(I) * C
  A(I+1) = A(I+1) + B(I+1) * C
  A(I+2) = A(I+2) + B(I+2) * C
  A(I+3) = A(I+3) + B(I+3) * C
ENDDO
```

El número de iteraciones requeridas en el ciclo precondicionante es el residuo de dividir el número total de iteraciones entre la cantidad de veces

que se desenrolló. Si, a tiempo de ejecución, N resulta ser divisible entre 4, no habrá iteraciones sobrantes, y el ciclo precondicionante no se ejecutará.

La ejecución especulativa en las arquitecturas post RISC puede reducir o eliminar la necesidad de desenrollar un bucle que operará sobre valores que deben recuperarse de la memoria principal. Dado que las operaciones de carga pueden tomar mucho tiempo relativo a los cálculos, el ciclo se desenrolla naturalmente. Mientras que el procesador está esperando a que finalice la primera carga, puede ejecutar especulativamente tres o cuatro iteraciones del ciclo adelante de la primera carga, desenrollando efectivamente el ciclo en el Buffer de Reordenamiento de Instrucciones.

Asumiendo un valor grande de N, el ciclo previo era un candidato ideal para desenrollarse. Las iteraciones pueden ejecutarse en cualquier orden, y las entrañas del ciclo eran pequeñas. Pero como tal vez sospeche, éste no siempre es el caso; ciertas clases de ciclos no pueden desenrollarse tan fácilmente. Adicionalmente, la forma en que se usa un ciclo cuando el programa se ejecuta puede descalificarlo para desenrollado de ciclos, incluso aunque parezca prometedor.

En esta sección vamos a discutir unas pocas categorías de ciclos que no son generalmente buenos candidatos para el desenrollado, y le daremos algunas ideas de qué puede usted hacer al respecto. Ya hablamos acerca de varias de ellas en el capítulo previo, pero nuevamente resultan relevantes.

Ciclos con Bajo Conteo de Repeticiones

Para resultar efectivo, el desenrollado de ciclos requiere que el original tenga un número bastante grande de iteracionesl. Para comprender por qué, visualicemos qué sucede si el conteo total de iteraciones es bajo, tal vez menor a 10, o incluso menor que 4. Con un conteo de repeticiones tan bajo, el ciclo precondicionante está llevando a cabo una cantidad proporcionalmente alta de trabajo. Y no se supone que deba ser así, pues éste debe hacerse cargo de las pocas iteraciones sobrantes olvidadas por el ciclo principal desenrollado. Sin embargo, cuando el conteo de repeticiones es bajo, puede que esté usted haciendo una o dos pasadas por el ciclo desenrollado, mas una o dos pasadas a través del ciclo precondicionante. En otras palabras, tiene usted entre manos un mayor desorden; simplemente, el ciclo no debió haberse desenrollado.

Probablemente, la única ocasión en que tiene sentido desenrollar un ciclo con un conteo de repeticiones bajo, es cuando el número de iteraciones es constante y se conoce a tiempo de compilación. Por ejemplo, suponga que tiene el siguiente ciclo:

```
PARAMETER (NITER = 3)
DO I=1,NITER
  A(I) = B(I) * C
ENDDO
```

Dado que `NITER` vale constantemente 3, puede usted desenrollar de forma segura hasta una profundidad de 3 sin preocuparse por el ciclo precondicionante. De hecho, puede desechar completamente la estructura del ciclo y quedarse sólo con el ciclo desenrollado en el interior:

```
PARAMETER (NITER = 3)
A(1) = B(1) * C
A(2) = B(2) * C
A(3) = A(3) * C
```

Por supuesto, si el contador del número de repeticiones es bajo, probablemente no contribuya significativamente al tiempo de ejecución global, a menos que se trate de un ciclo en el centro de otro más grande. Entonces probablemente quiera usted desenrollarlo completamente, o dejarlo solo.

Ciclos Gruesos

El desenrollado de ciclos ayuda al rendimiento porque engruesa cada uno con más cálculos por iteración. Por la misma razón, si un ciclo particular ya es grueso, desenrollarlo no será de mucha ayuda, pues su sobrecarga ya está distribuida sobre un número grande de instrucciones. De hecho, desenrollar un ciclo grueso puede incluso volver más lento su programa, porque incrementará el tamaño del segmento de texto, colocando una carga adicional sobre el sistema de memoria (lo explicaremos con más detalle un poco más adelante). Una buena regla empírica consiste en ir en busca de

rendimiento en cualquier otra parte cuando el ciclo interno excede tres o cuatro sentencias.

Ciclos que Contienen Llamados a Procedimientos

Tal como sucede con los ciclos gruesos, aquellos que contienen llamados a subrutinas o funciones generalmente no son buenos candidatos para el desenrollado, debido a varias razones. Primero, a menudo ya contienen un buen número de instrucciones. Y si la subrutina que se está invocando es gruesa, también engruesa al ciclo que la invoca. El tamaño del ciclo puede no parecerlo cuando usted lo revisa, pero la invocación a la función puede ocultar muchas más instrucciones.

Segundo, cuando la rutina invocadora y la subrutina se compilan por separado, es imposible que el compilador entremezcle las instrucciones. Un ciclo que se desenrolla en una serie de llamados a funciones se comporta de forma muy parecida a como lo hacía el ciclo original antes de desenrollarse.

Por último, la sobrecarga por invocar la función es cara: deben salvarse los valores de los registros; debe prepararse la lista de argumentos. El tiempo gastado en invocar a y retornar de una subrutina puede ser mucho mayor que el provocado por el del ciclo. Desenrollar el ciclo para amortizar su costo entre varias invocaciones no resulta lo suficiente como para justificar el esfuerzo.

La regla general cuando intervienen procedimientos, es primero tratar de eliminarlos durante la fase de "remoción de desorden", y cuando dicha fase ha concluido, comprobar si el desenrollado proporciona una mejora de rendimiento adicional.

Ciclos con Bifurcaciones en su Interior

[link] mostramos cómo eliminar ciertos tipos de bifurcaciones, pero por supuesto no podemos deshacernos de todas. En el caso de aquéllas independientes de la iteración, podemos obtener cierto beneficio de desenrollar el ciclo. La condición de la sentencia IF se vuelve parte de las operaciones que deben contarse para determinar el valor del desenrollado

del ciclo. Abajo presentamos un ciclo doblemente anidado. El interno prueba el valor de `B(J,I)`:

```
      DO I=1,N
        DO J=1,N
          IF (B(J,I) .GT. 1.0) A(J,I) = A(J,I) +
B(J,I) * C
        ENDDO
      ENDDO
```

Cada iteración es independiente de cualesquiera otra, así que desenrollarlo no representa un problema. Sólo debemos mantener el ciclo externo inalterado:

```
      II = IMOD (N,4)
      DO I=1,N
        DO J=1,II
          IF (B(J,I) .GT. 1.0)
     +      A(J,I) = A(J,I) + B(J,I) * C
          ENDDO
          DO J=II+1,N,4
            IF (B(J,I) .GT. 1.0)
     +        A(J,I) = A(J,I) + B(J,I) * C
            IF (B(J+1,I) .GT. 1.0)
     +        A(J+1,I) = A(J+1,I) + B(J+1,I) * C
            IF (B(J+2,I) .GT. 1.0)
     +        A(J+2,I) = A(J+2,I) + B(J+2,I) * C
            IF (B(J+3,I) .GT. 1.0)
     +        A(J+3,I) = A(J+3,I) + B(J+3,I) * C
          ENDDO
        ENDDO
```

Este enfoque funciona particularmente bien si el procesador que está usted usando soporta ejecución condicional. Como se describió con anterioridad, la ejecución condicional puede reemplazar una bifurcación y una operación, por una única sentencia ejecutada condicionalmente. En procesadores superescalares con ejecución condicional, tales ciclos desenrollados se ejecutan muy bien.

Nested Loops

When you embed loops within other loops, you create a *loop nest*. The loop or loops in the center are called the *inner* loops. The surrounding loops are called *outer* loops. Depending on the construction of the loop nest, we may have some flexibility in the ordering of the loops. At times, we can swap the outer and inner loops with great benefit. In the next sections we look at some common loop nestings and the optimizations that can be performed on these loop nests.

Often when we are working with nests of loops, we are working with multidimensional arrays. Computing in multidimensional arrays can lead to non-unit-stride memory access. Many of the optimizations we perform on loop nests are meant to improve the memory access patterns.

First, we examine the computation-related optimizations followed by the memory optimizations.

Outer Loop Unrolling

If you are faced with a loop nest, one simple approach is to unroll the inner loop. Unrolling the innermost loop in a nest isn't any different from what we saw above. You just pretend the rest of the loop nest doesn't exist and approach it in the nor- mal way. However, there are times when you want to apply loop unrolling not just to the inner loop, but to outer loops as well — or perhaps only to the outer loops. Here's a typical loop nest:

```
      for (i=0; i<n; i++)
          for (j=0; j<n; j++)
              for (k=0; k<n; k++)
                  a[i][j][k] = a[i][j][k] + b[i]
[j][k] * c;
```

To unroll an outer loop, you pick one of the outer loop index variables and replicate the innermost loop body so that several iterations are performed at the same time, just like we saw in the [link]. The difference is in the index variable for which you unroll. In the code below, we have unrolled the middle `(j)` loop twice:

```
      for (i=0; i<n; i++)
          for (j=0; j<n; j+=2)
              for (k=0; k<n; k++) {
                  a[i][j][k] = a[i][j][k] + b[i]
[k][j] * c;
                  a[i][j+1][k] = a[i][j+1][k] +
b[i][k][j+1] * c;
              }
```

We left the `k` loop untouched; however, we could unroll that one, too. That would give us outer *and* inner loop unrolling at the same time:

```
      for (i=0; i<n; i++)
          for (j=0; j<n; j+=2)
              for (k=0; k<n; k+=2) {
                  a[i][j][k]     = a[i][j][k]
+ b[i][k][j] * c;
                  a[i][j+1][k]   = a[i][j+1][k]
+ b[i][k][j+1] * c;
                  a[i][j][k+1]   = a[i][j][k+1]
+ b[i][k+1][j] * c;
                  a[i][j+1][k+1] = a[i][j+1][k+1]
+ b[i][k+1][j+1] * c;
              }
```

We could even unroll the `i` loop too, leaving eight copies of the loop innards. (Notice that we completely ignored preconditioning; in a real application, of course, we couldn't.)

Outer Loop Unrolling to Expose Computations

Say that you have a doubly nested loop and that the inner loop trip count is low — perhaps 4 or 5 on average. Inner loop unrolling doesn't make sense in this case because there won't be enough iterations to justify the cost of the preconditioning loop. However, you may be able to unroll an outer loop. Consider this loop, assuming that `M` is small and `N` is large:

```
DO I=1,N
  DO J=1,M
    A(J,I) = B(J,I) + C(J,I) * D
  ENDDO
ENDDO
```

Unrolling the `I` loop gives you lots of floating-point operations that can be overlapped:

```
II = IMOD (N,4)
DO I=1,II
  DO J=1,M
    A(J,I) = B(J,I) + C(J,I) * D
  ENDDO
ENDDO

DO I=II,N,4
  DO J=1,M
    A(J,I)   = B(J,I)   + C(J,I) * D
    A(J,I+1) = B(J,I+1) + C(J,I+1) * D
```

```
    A(J,I+2) = B(J,I+2) + C(J,I+2) * D
    A(J,I+3) = B(J,I+3) + C(J,I+3) * D
  ENDDO
ENDDO
```

In this particular case, there is bad news to go with the good news: unrolling the outer loop causes strided memory references on `A`, `B`, and `C`. However, it probably won't be too much of a problem because the inner loop trip count is small, so it naturally groups references to conserve cache entries.

Outer loop unrolling can also be helpful when you have a nest with recursion in the inner loop, but not in the outer loops. In this next example, there is a first- order linear recursion in the inner loop:

```
DO J=1,M
  DO I=2,N
    A(I,J) = A(I,J) + A(I-1,J) * B
  ENDDO
ENDDO
```

Because of the recursion, we can't unroll the inner loop, but we can work on several copies of the outer loop at the same time. When unrolled, it looks like this:

```
JJ = IMOD (M,4)
DO J=1,JJ
  DO I=2,N
    A(I,J) = A(I,J) + A(I-1,J) * B
  ENDDO
ENDDO

DO J=1+JJ,M,4
```

```
    DO I=2,N
      A(I,J)   = A(I,J)   + A(I-1,J)   * B
      A(I,J+1) = A(I,J+1) + A(I-1,J+1) * B
      A(I,J+2) = A(I,J+2) + A(I-1,J+2) * B
      A(I,J+3) = A(I,J+3) + A(I-1,J+3) * B
    ENDDO
  ENDDO
```

You can see the recursion still exists in the `I` loop, but we have succeeded in finding lots of work to do anyway.

Sometimes the reason for unrolling the outer loop is to get a hold of much larger chunks of things that can be done in parallel. If the outer loop iterations are independent, and the inner loop trip count is high, then each outer loop iteration represents a significant, parallel chunk of work. On a single CPU that doesn’t matter much, but on a tightly coupled multiprocessor, it can translate into a tremendous increase in speeds.

Loop Interchange

Loop interchange is a technique for rearranging a loop nest so that the right stuff is at the center. What the right stuff is depends upon what you are trying to accomplish. In many situations, loop interchange also lets you swap high trip count loops for low trip count loops, so that activity gets pulled into the center of the loop nest.[footnote]
It's also good for improving memory access patterns.

Loop Interchange to Move Computations to the Center

When someone writes a program that represents some kind of real-world model, they often structure the code in terms of the model. This makes perfect sense. The computer is an analysis tool; you aren't writing the code on the computer's behalf. However, a model expressed naturally often works on one point in space at a time, which tends to give you insignificant inner loops — at least in terms of the trip count. For performance, you might want to interchange inner and outer loops to pull the activity into the center, where you can then do some unrolling. Let's illustrate with an example. Here's a loop where `KDIM` time-dependent quantities for points in a two-dimensional mesh are being updated:

```
      PARAMETER (IDIM = 1000, JDIM = 1000, KDIM =
3)
        ...
      DO I=1,IDIM
        DO J=1,JDIM
          DO K=1,KDIM
            D(K,J,I) = D(K,J,I) + V(K,J,I) * DT
          ENDDO
        ENDDO
      ENDDO
```

In practice, `KDIM` is probably equal to 2 or 3, where `J` or `I`, representing the number of points, may be in the thousands. The way it is written, the inner loop has a very low trip count, making it a poor candidate for unrolling.

By interchanging the loops, you update one quantity at a time, across all of the points. For tuning purposes, this moves larger trip counts into the inner loop and allows you to do some strategic unrolling:

```
DO K=1,KDIM
  DO J=1,JDIM
    DO I=1,IDIM
      D(K,J,I) = D(K,J,I) + V(K,J,I) * DT
    ENDDO
  ENDDO
ENDDO
```

This example is straightforward; it's easy to see that there are no inter-iteration dependencies. But how can you tell, in general, when two loops can be inter- changed? Interchanging loops might violate some dependency, or worse, only violate it occasionally, meaning you might not catch it when optimizing. Can we interchange the loops below?

```
DO I=1,N-1
  DO J=2,N
    A(I,J) = A(I+1,J-1) * B(I,J)
    C(I,J) = B(J,I)
  ENDDO
ENDDO
```

While it is possible to examine the loops by hand and determine the dependencies, it is much better if the compiler can make the determination.

Very few single-processor compilers automatically perform loop interchange. However, the compilers for high-end vector and parallel computers generally interchange loops if there is some benefit and if interchanging the loops won't alter the program results.[footnote]
When the compiler performs automatic parallel optimization, it prefers to run the outermost loop in parallel to minimize overhead and unroll the innermost loop to make best use of a superscalar or vector processor. For this reason, the compiler needs to have some flexibility in ordering the loops in a loop nest.

Memory Access Patterns

The best pattern is the most straightforward: increasing and unit sequential. For an array with a single dimension, stepping through one element at a time will accomplish this. For multiply-dimensioned arrays, access is fastest if you iterate on the array subscript offering the smallest *stride* or step size. In FORTRAN programs, this is the leftmost subscript; in C, it is the rightmost. The FORTRAN loop below has unit stride, and therefore will run quickly:

```
DO J=1,N
  DO I=1,N
    A(I,J) = B(I,J) + C(I,J) * D
  ENDDO
ENDDO
```

In contrast, the next loop is slower because its stride is `N` (which, we assume, is greater than 1). As N increases from one to the length of the cache line (adjusting for the length of each element), the performance worsens. Once `N` is longer than the length of the cache line (again adjusted for element size), the performance won't decrease:

```
DO J=1,N
  DO I=1,N
    A(J,I) = B(J,I) + C(J,I) * D
  ENDDO
ENDDO
```

Here's a unit-stride loop like the previous one, but written in C:

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        a[i][j] = a[i][j] + c[i][j] * d;
```

Unit stride gives you the best performance because it conserves cache entries. Recall how a data cache works.[footnote] Your program makes a memory reference; if the data is in the cache, it gets returned immediately. If not, your program suffers a cache miss while a new cache line is fetched from main memory, replacing an old one. The line holds the values taken from a handful of neighboring memory locations, including the one that caused the cache miss. If you loaded a cache line, took one piece of data from it, and threw the rest away, you would be wasting a lot of time and memory bandwidth. However, if you brought a line into the cache and consumed everything in it, you would benefit from a large number of memory references for a small number of cache misses. This is exactly what you get when your program makes unit-stride memory references. See [link].

The worst-case patterns are those that jump through memory, especially a large amount of memory, and particularly those that do so without apparent rhyme or reason (viewed from the outside). On jobs that operate on very large data structures, you pay a penalty not only for cache misses, but for TLB misses too.[footnote] It would be nice to be able to rein these jobs in so that they make better use of memory. Of course, you can't eliminate memory references; programs have to get to their data one way or another. The question is, then: how can we restructure memory access patterns for the best performance?
The Translation Lookaside Buffer (TLB) is a cache of translations from virtual memory addresses to physical memory addresses. For more information, refer back to [link].

In the next few sections, we are going to look at some tricks for restructuring loops with strided, albeit predictable, access patterns. The tricks will be familiar; they are mostly loop optimizations from [link], used here for different reasons. The underlying goal is to minimize cache and

TLB misses as much as possible. You will see that we can do quite a lot, although some of this is going to be ugly.

Loop Interchange to Ease Memory Access Patterns

Loop interchange is a good technique for lessening the impact of strided memory references. Let's revisit our FORTRAN loop with non-unit stride. The good news is that we can easily interchange the loops; each iteration is independent of every other:

```
DO J=1,N
  DO I=1,N
    A(J,I) = B(J,I) + C(J,I) * D
  ENDDO
ENDDO
```

After interchange, `A`, `B`, and `C` are referenced with the leftmost subscript varying most quickly. This modification can make an important difference in performance. We traded three N-strided memory references for unit strides:

```
DO I=1,N
  DO J=1,N
    A(J,I) = B(J,I) + C(J,I) * D
  ENDDO
ENDDO
```

Matrix Multiplication

Matrix multiplication is a common operation we can use to explore the options that are available in optimizing a loop nest. A programmer who has

just finished reading a linear algebra textbook would probably write matrix multiply as it appears in the example below:

```
DO I=1,N
  DO J=1,N
    SUM = 0
    DO K=1,N
      SUM = SUM + A(I,K) * B(K,J)
    ENDDO
    C(I,J) = SUM
  ENDDO
ENDDO
```

The problem with this loop is that the `A(I,K)` will be non-unit stride. Each iteration in the inner loop consists of two loads (one non-unit stride), a multiplication, and an addition.

Given the nature of the matrix multiplication, it might appear that you can't eliminate the non-unit stride. However, with a simple rewrite of the loops all the memory accesses can be made unit stride:

```
DO J=1,N
  DO I=1,N
    C(I,J) = 0.0
  ENDDO
ENDDO

DO K=1,N
  DO J=1,N
    SCALE = B(K,J)
    DO I=1,N
      C(I,J) = C(I,J) + A(I,K) * SCALE
    ENDDO
```

```
    ENDDO
  ENDDO
```

Now, the inner loop accesses memory using unit stride. Each iteration performs two loads, one store, a multiplication, and an addition. When comparing this to the previous loop, the non-unit stride loads have been eliminated, but there is an additional store operation. Assuming that we are operating on a cache-based system, and the matrix is larger than the cache, this extra store won't add much to the execution time. The store is to the location in `C(I,J)` that was used in the load. In most cases, the store is to a line that is already in the in the cache. The `B(K,J)` becomes a constant scaling factor within the inner loop.

When Interchange Won't Work

In the matrix multiplication code, we encountered a non-unit stride and were able to eliminate it with a quick interchange of the loops. Unfortunately, life is rarely this simple. Often you find some mix of variables with unit and non-unit strides, in which case interchanging the loops moves the damage around, but doesn't make it go away.

The loop to perform a matrix transpose represents a simple example of this dilemma:

```
      DO I=1,N                                  DO 20
J=1,M
        DO J=1,M                                  DO 10
I=1,N
          A(J,I) = B(I,J)
A(J,I) = B(I,J)
        ENDDO                                     ENDDO
      ENDDO                                     ENDDO
```

Whichever way you interchange them, you will break the memory access pattern for either A or B. Even more interesting, you have to make a choice between strided loads vs. strided stores: which will it be?[footnote] We really need a general method for improving the memory access patterns for *both* A and B, not one or the other. We'll show you such a method in [link]. I can't tell you which is the better way to cast it; it depends on the brand of computer. Some perform better with the loops left as they are, sometimes by more than a factor of two. Others perform better with them interchanged. The difference is in the way the processor handles updates of main memory from cache.

Blocking to Ease Memory Access Patterns

Blocking is another kind of memory reference optimization. As with loop interchange, the challenge is to retrieve as much data as possible with as few cache misses as possible. We'd like to rearrange the loop nest so that it works on data in little neighborhoods, rather than striding through memory like a man on stilts. Given the following vector sum, how can we rearrange the loop?

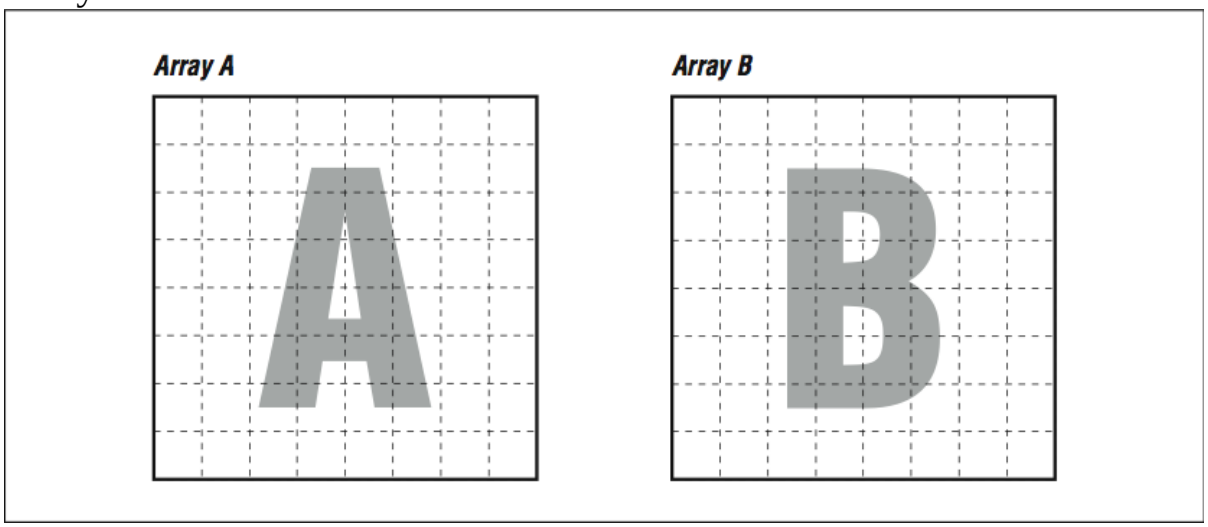
```
DO I=1,N
  DO J=1,N
    A(J,I) = A(J,I) + B(I,J)
  ENDDO
ENDDO
```

This loop involves two vectors. One is referenced with unit stride, the other with a stride of N. We can interchange the loops, but one way or another we still have `N`-strided array references on either `A` or `B`, either of which is undesirable. The trick is to *block* references so that you grab a few elements of `A`, and then a few of `B`, and then a few of `A`, and so on — in neighborhoods. We make this happen by combining inner and outer loop unrolling:

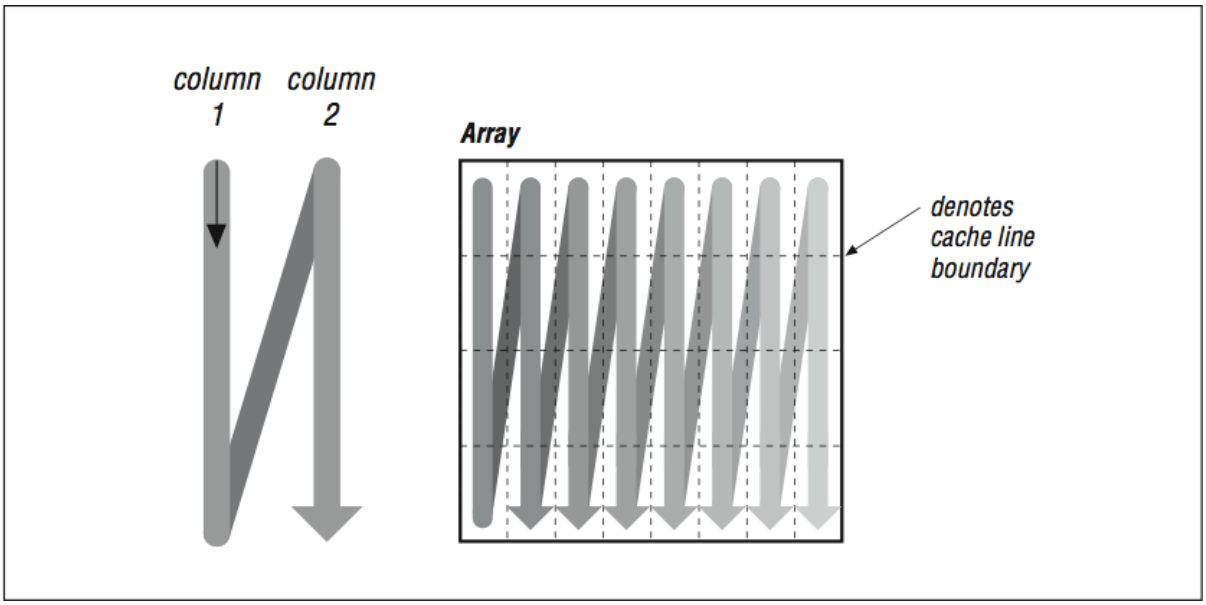
```
DO I=1,N,2
  DO J=1,N,2
    A(J,I)     = A(J,I)     + B(I,J)
    A(J+1,I)   = A(J+1,I)   + B(I,J+1)
    A(J,I+1)   = A(J,I+1)   + B(I+1,J)
    A(J+1,I+1) = A(J+1,I+1) + B(I+1,J+1)
  ENDDO
ENDDO
```

Use your imagination so we can show why this helps. Usually, when we think of a two-dimensional array, we think of a rectangle or a square (see [link]). Remember, to make programming easier, the compiler provides the illusion that two-dimensional arrays A and B are rectangular plots of memory as in [link]. Actually, memory is sequential storage. In FORTRAN, a two-dimensional array is constructed in memory by logically lining memory “strips” up against each other, like the pickets of a cedar fence. (It’s the other way around in C: rows are stacked on top of one another.) Array storage starts at the upper left, proceeds down to the bottom, and then starts over at the top of the next column. Stepping through the array with unit stride traces out the shape of a backwards “N,” repeated over and over, moving to the right.

Arrays A and B



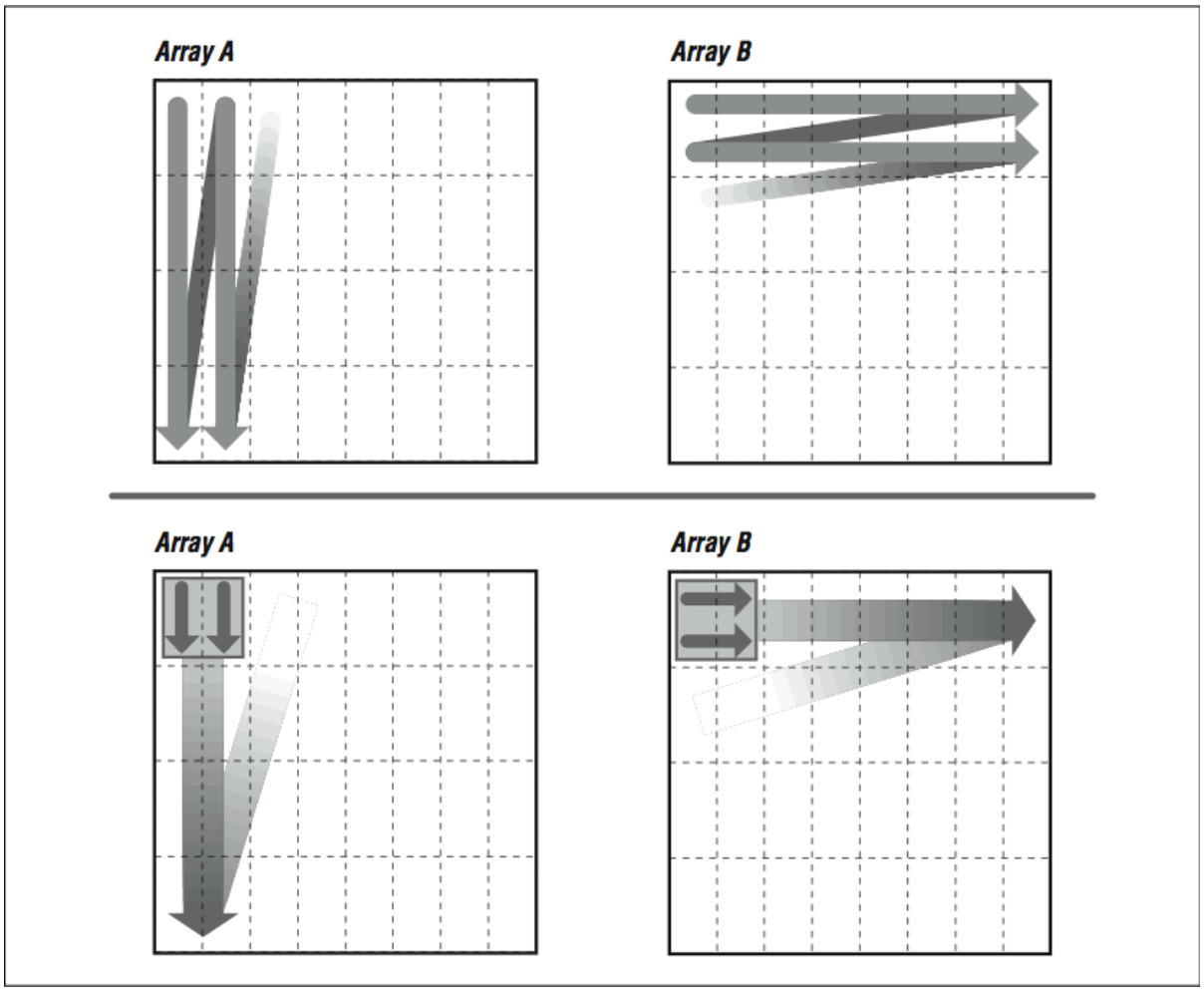
How array elements are stored



Imagine that the thin horizontal lines of [link] cut memory storage into pieces the size of individual cache entries. Picture how the loop will traverse them. Because of their index expressions, references to `A` go from top to bottom (in the backwards "N" shape), consuming every bit of each cache line, but references to `B` dash off to the right, using one piece of each cache entry and discarding the rest (see [link], top). This low usage of cache entries will result in a high number of cache misses.

If we could somehow rearrange the loop so that it consumed the arrays in small rectangles, rather than strips, we could conserve some of the cache entries that are being discarded. This is exactly what we accomplished by unrolling both the inner and outer loops, as in the following example. Array `A` is referenced in several strips side by side, from top to bottom, while `B` is referenced in several strips side by side, from left to right (see [link], bottom). This improves cache performance and lowers runtime.

For really big problems, more than cache entries are at stake. On virtual memory machines, memory references have to be translated through a `TLB`. If you are dealing with large arrays, `TLB` misses, in addition to cache misses, are going to add to your runtime.
2×2 squares



Here's something that may surprise you. In the code below, we rewrite this loop yet again, this time blocking references at two different levels: in `2×2` squares to save cache entries, and by cutting the original loop in two parts to save `TLB` entries:

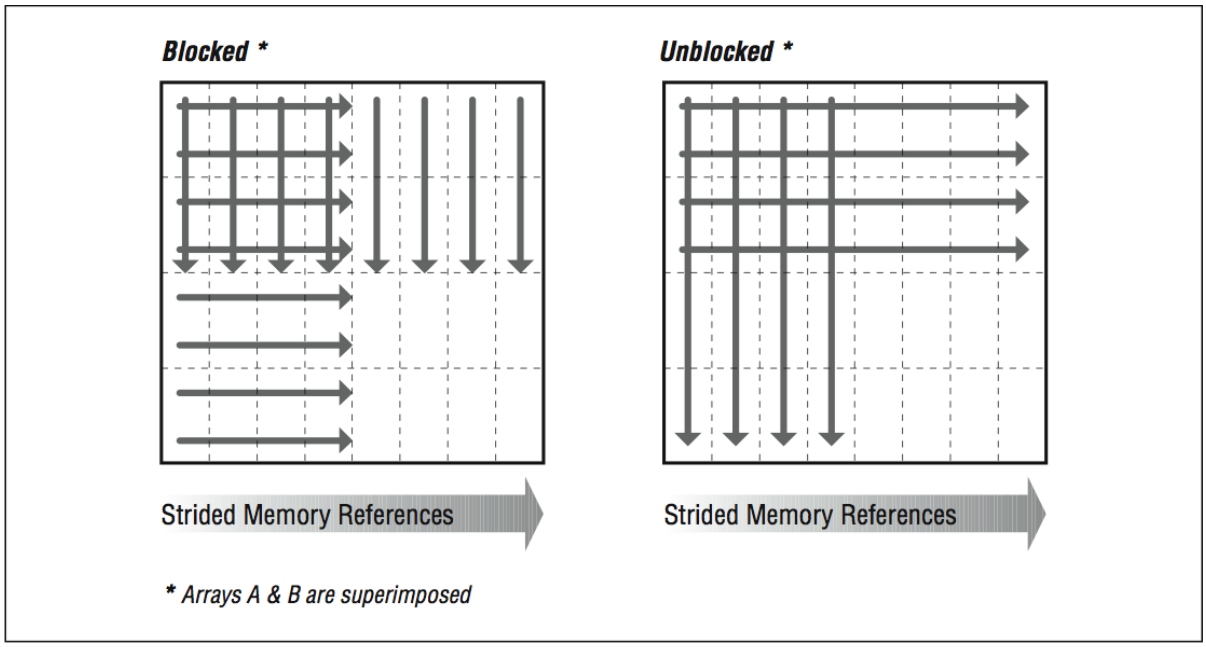
```
DO I=1,N,2
  DO J=1,N/2,2
    A(J,I) = A(J,I) + B(I,J)
    A(J+1,I) = A(J+1,I) + B(I+1,J)
    A(J,I+1) = A(J,I+1) + B(I+1,J)
    A(J+1,I+1) = A(J+1,I+1) + B(I+1,J+1)
  ENDDO
ENDDO
```

```
DO I=1,N,2
  DO J=N/2+1,N,2
    A(J,I) = A(J,I) + B(I,J)
    A(J+1,I) = A(J+1,I) + B(I+1,J)
    A(J,I+1) = A(J,I+1) + B(I+1,J)
    A(J+1,I+1) = A(J+1,I+1) + B(I+1,J+1)
  ENDDO
ENDDO
```

You might guess that adding more loops would be the wrong thing to do. But if you work with a reasonably large value of `N`, say 512, you will see a significant increase in performance. This is because the two arrays `A` and `B` are each 256 KB × 8 bytes = 2 MB when `N` is equal to 512 — larger than can be handled by the TLBs and caches of most processors.

The two boxes in [link] illustrate how the first few references to A and B look superimposed upon one another in the blocked and unblocked cases. Unblocked references to B zing off through memory, eating through cache and TLB entries. Blocked references are more sparing with the memory system.

Picture of unblocked versus blocked references



You can take blocking even further for larger problems. This code shows another method that limits the size of the inner loop and visits it repeatedly:

```
II = MOD (N,16)
JJ = MOD (N,4)

DO I=1,N
  DO J=1,JJ
    A(J,I) = A(J,I) + B(J,I)
  ENDDO
ENDDO

DO I=1,II
  DO J=JJ+1,N
      A(J,I) = A(J,I) + B(J,I)
      A(J,I) = A(J,I) + 1.0D0
  ENDDO
ENDDO

DO I=II+1,N,16
  DO J=JJ+1,N,4
    DO K=I,I+15
      A(J,K)   = A(J,K)   + B(K,J)
      A(J+1,K) = A(J+1,K) + B(K,J+1)
      A(J+2,K) = A(J+2,K) + B(K,J+2)
      A(J+3,K) = A(J+3,K) + B(K,J+3)
    ENDDO
  ENDDO
ENDDO
```

Where the inner `I` loop used to execute `N` iterations at a time, the new `K` loop executes only 16 iterations. This divides and conquers a large memory address space by cutting it into little pieces.

While these blocking techniques begin to have diminishing returns on single-processor systems, on large multiprocessor systems with nonuniform memory access (NUMA), there can be significant benefit in carefully arranging memory accesses to maximize reuse of both cache lines and main memory pages.

Again, the combined unrolling and blocking techniques we just showed you are for loops with *mixed stride* expressions. They work very well for loop nests like the one we have been looking at. However, if all array references are strided the same way, you will want to try loop unrolling or loop interchange first.

Programs That Require More Memory Than You Have

People occasionally have programs whose memory size requirements are so great that the data can't fit in memory all at once. At any time, some of the data has to reside outside of main memory on secondary (usually disk) storage. These out-of- core solutions fall into two categories:

- Software-managed, out-of-core solutions
- Virtual memory–managed, out-of-core solutions

With a software-managed approach, the programmer has recognized that the problem is too big and has modified the source code to move sections of the data out to disk for retrieval at a later time. The other method depends on the computer's memory system handling the secondary storage requirements on its own, some- times at a great cost in runtime.

Software-Managed, Out-of-Core Solutions

Most codes with software-managed, out-of-core solutions have adjustments; you can tell the program how much memory it has to work with, and it takes care of the rest. It is important to make sure the adjustment is set correctly. Code that was tuned for a machine with limited memory could have been ported to another without taking into account the storage available. Perhaps the whole problem will fit easily.

If we are writing an out-of-core solution, the trick is to group memory references together so that they are localized. This usually occurs naturally as a side effect of partitioning, say, a matrix factorization into groups of columns. Blocking references the way we did in the previous section also corrals memory references together so you can treat them as memory "pages." Knowing when to ship them off to disk entails being closely involved with what the program is doing.

Closing Notes

Loops are the heart of nearly all high performance programs. The first goal with loops is to express them as simply and clearly as possible (i.e., eliminates the clutter). Then, use the profiling and timing tools to figure out which routines and loops are taking the time. Once you find the loops that are using the most time, try to determine if the performance of the loops can be improved.

First try simple modifications to the loops that don’t reduce the clarity of the code. You can also experiment with compiler options that control loop optimizations. Once you’ve exhausted the options of keeping the code looking clean, and if you still need more performance, resort to hand-modifying to the code. Typically the loops that need a little hand-coaxing are loops that are making bad use of the memory architecture on a cache-based system. Hopefully the loops you end up changing are only a few of the overall loops in the program.

However, before going too far optimizing on a single processor machine, take a look at how the program executes on a parallel system. Sometimes the modifications that improve performance on a single-processor system confuses the parallel-processor compiler. The compilers on parallel and vector systems generally have more powerful optimization capabilities, as they must identify areas of your code that will execute well on their specialized hardware. These compilers have been interchanging and unrolling loops automatically for some time now.

Exercises

Exercise:

Problem:

Why is an unrolling amount of three or four iterations generally sufficient for simple vector loops on a RISC processor? What relationship does the unrolling amount have to floating-point pipeline depths?

Exercise:

Problem:

On a processor that can execute one floating-point multiply, one floating-point addition/subtraction, and one memory reference per cycle, what's the best performance you could expect from the following loop?

```
DO I = 1,10000
  A(I) = B(I) * C(I) - D(I) * E(I)
ENDDO
```

Exercise:

Problem:

Try unrolling, interchanging, or blocking the loop in subroutine `BAZFAZ` to increase the performance. What method or combination of methods works best? Look at the assembly language created by the compiler to see what its approach is at the highest level of optimization.

Note: Compile the main routine and `BAZFAZ` separately; adjust `NTIMES` so that the untuned run takes about one minute; and use the

compiler's default optimization level.

```
            PROGRAM MAIN
            IMPLICIT NONE
            INTEGER M,N,I,J
            PARAMETER (N = 512, M = 640, NTIMES
= 500)
            DOUBLE PRECISION Q(N,M), R(M,N)
      C
            DO I=1,M
              DO J=1,N
                Q(J,I) = 1.0D0
                R(I,J) = 1.0D0
              ENDDO
            ENDDO
      C
            DO I=1,NTIMES
              CALL BAZFAZ (Q,R,N,M)
            ENDDO
            END

            SUBROUTINE BAZFAZ (Q,R,N,M)
            IMPLICIT NONE
            INTEGER M,N,I,J
            DOUBLE PRECISION Q(N,M), R(N,M)
      C
            DO I=1,N
              DO J=1,M
                R(I,J) = Q(I,J) * R(J,I)
              ENDDO
            ENDDO
      C
```

```
END
```

Exercise:

Problem:

Code the matrix multiplication algorithm in the “straightforward” manner and compile it with various optimization levels. See if the compiler performs any type of loop interchange.

Try the same experiment with the following code:

```
DO I=1,N
  DO J=1,N
    A(I,J) = A(I,J) + 1.3
  ENDDO
ENDDO
```

Do you see a difference in the compiler’s ability to optimize these two loops? If you see a difference, explain it.

Exercise:

Problem:

Code the matrix multiplication algorithm both the ways shown in this chapter. Execute the program for a range of values for N. Graph the execution time divided by N3 for values of N ranging from 50×50 to 500×500. Explain the performance you see.

Entendiendo el Paralelismo - Introducción

En cierto sentido, hemos estado hablando acerca de paralelismo desde el inicio del libro. Sólo que en vez de llamarlo "paralelismo", hemos usado términos como "entubamiento", "superescalar" y "flexibilidad del compilador". Conforme nos adentramos en la programación de multiprocesadores, debemos incrementar nuestro entendimiento del paralelismo, para poder comprender cómo programar estos sistemas de forma efectiva. En corto, mientras obtenemos más recursos paralelos, necesitamos encontrar más paralelismo en nuestro código.

Cuando hablamos de paralelismo, necesitamos entender el concepto de granularidad. La granularidad del paralelismo indica el tamaño de los cálculos que se están realizando simultáneamente entre sincronizaciones. Algunos ejemplos de paralelismo, ordenados de acuerdo al tamaño de grano, son:

- Cuando realiza una suma de enteros de 32 bits, usando un sumador con acarreo por desplazamiento, puede usted sumar parcialmente los bits 0 y 1 al mismo tiempo que los bits 2 y 3.
- En un procesador con entubamiento, mientras se decodifica una instrucción, puede recuperarse de la memoria la siguiente.
- En un procesador escalar de dos vías puede ejecutarse, en un solo ciclo, cualquier combinación de una instrucción entera y de punto flotante.
- En un multiprocesador, puede usted dividir las iteraciones de un ciclo entre los cuatro procesadores del sistema.
- Puede usted dividir un arreglo grande entre cuatro estaciones de trabajo conectadas en red. Cada estación puede operar sobre su propia información local, y luego intercambiar los valores de frontera al final de cada paso de tiempo.

En este capítulo, comenzamos con el *paralelismo a nivel de instrucciones* (entubamiento y supesescalar) y avanzamos hacia el *paralelismo a nivel de hilos de ejecución*, que es el que necesitamos para los sistemas multiprocesador. Es importante señalar que estos niveles distintos de paralelismo generalmente no entran en conflicto. Incrementar el paralelismo

a nivel de hilos en problemas de grano grueso, a menudo hace que salga a flote el paralelismo de grano más fino.

El siguiente ciclo está lleno de paralelismo:

```
DO I=1,16000
  A(I) = B(I) * 3.14159
ENDDO
```

Hemos expresado el ciclo en una forma que parece implicar que debe calcularse A(1) primero, seguido por A(2), y así sucesivamente. Sin embargo, una vez completado el ciclo, no tiene importancia si A(16000) se calculó antes o después que A(15999). Bien pudiera haberse calculado primero todos los valores pares de I y luego los nones. Tampoco hubiera hecho una diferencia si las 16,000 iteraciones se hubieran calculado simultáneamente, usando un procesador superescalar de 16,000 vías. [footnote] Si el compilador es flexible en cuanto al orden en que ejecuta las instrucciones que componen el programa, puede realizar todas estas simultáneamente siempre que haya disponible hardware paralelo. Curiosamente, esta idea no es tan descabellada como pareciera. Sobre una computadora SIMD (una sola instrucción, múltiples datos) como la Connection CM-2 con 16,384 procesadores, tomaría tres ciclos de instrucción procesar este ciclo completo.

Una técnica usada por los científicos computacionales para analizar formalmente el paralelismo potencial de un algoritmo, consiste en caracterizar cuan rápidamente se ejecutaría con un procesador superescalar de "infinito número de vías".

No todos los ciclos contienen tanto paralelismo como este sencillo que analizamos. Necesitamos identificar aquellas cosas que limitan el paralelismo en nuestro código, y quitarlas siempre que sea posible. En capítulos previos hemos identificado y retirado el desorden y reescrito los ciclos para simplificarlos.

Este capítulo también proporciona [link], en muchas formas. Revisaremos la mecánica de la compilación de código, la cuál se aplica aquí, pero no daremos respuesta a todos los "porqués". Las técnicas básicas de análisis de bloques forman la base del trabajo que realiza el compilador cuando trata de lograr mayor paralelismo. Al observar dos piezas de datos, instrucciones, o datos e instrucciones, un compilador debe responder la pregunta: "¿dependen las unas de las otras?" Existen tres respuesta posibles: sí, no, y no lo sabemos. La tercera respuesta es, en la práctica, la misma que sí, porque el compilador debe comportarse conservadoramente cuando no puede garantizar que sea seguro reordenar las instrucciones.

Ayudar al compilador a reconocer el paralelismo es uno de los enfoques básicos que toman los especialistas para afinar el código. Una ligera reescritura de un ciclo, o cierta información suplementaria dada al compilador, pueden convertir una respuesta de "no sabemos" en una oportunidad de lograr paralelismo. Es seguro que hay otras facetas en este proceso de afinación, tales como optimizar los patrones de acceso a memoria de forma que se adapten mejor al hardware, o reelaborar un algoritmo. Y no existe una sola forma correcta de hacerlo para todos los problemas; cualquier esfuerzo de mejora debe involucrar una combinación de técnicas.

Entendiendo el Paralelismo - Dependencias

Imagine una orquesta sinfónica donde cada músico toque sin tomar en cuenta al director o a lo otros músicos. Al primer movimiento de la batuta del director, cada músico recorre toda su partitura. Algunos finalizan mucho antes que otros, dejan el escenario y se van a casa. La cacofonía resultante no parece música (pensándolo bien, recuerdo algo al jazz experimental), porque carece por completo de coordinación. Por supuesto, esta no es la forma en que se interpreta la música. Un programa de computadora, como una pieza musical, se teje en un telar que lo despliega conforme avanza el tiempo (aunque tal vez lo teje menos compactamente). Ciertas cosas deben suceder antes que otras, y hay una tasa de ejecución para el proceso en su conjunto.

En los programas de computadora, cuando un evento A debe suceder antes de otro evento B, se dice que B es *dependiente* de A. Llamamos a esta relación entre ellos una dependencia. En ocasiones las dependencias se deben a cálculos o accesos a memoria, en cuyo caso les denominamos *dependencias de datos*. En otras ocasiones estamos a la espera de que ocurra un salto o la salida de un ciclo, en cuyo caso les denominamos *dependencias de control*. Cada tipo está presente en cada programa, sólo que en grados distintos. El objetivo es eliminar tantas de ellas como sea posible. Reacomodar el código, de forma que dos segmentos sean menos dependientes entre sí, expone el *paralelismo*, o las oportunidades de hacer muchas cosas a la vez.

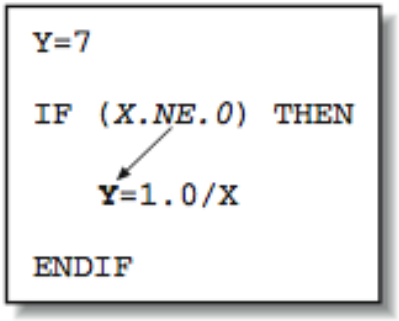
Dependencias de Control

Tal como la asignación de variables puede depender de otras asignaciones, el valor de una variable puede depender también del *flujo de control* del programa. Por ejemplo, una asignación adentro de una sentencia condicional sólo puede ocurrir si la condición se evalúa como verdadera. Lo mismo puede decirse de una asignación al interior de un ciclo; si nunca se entra en dicho ciclo, ninguna sentencia en su interior se ejecutará.

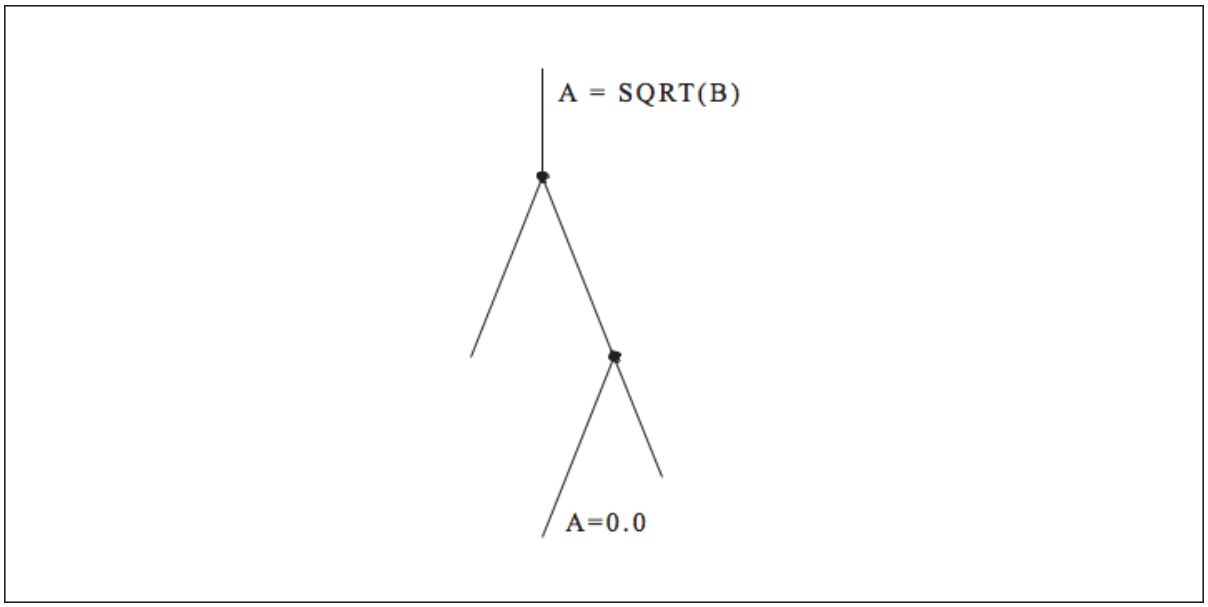
Cuando los cálculos ocurren como consecuencia del flujo de control, decimos que se trata de una *dependencia de control*, tal como en el código

que aparece más abajo y que se explica gráficamente en [link]. La asignación ubicada dentro del bloque condicional puede o no ejecutarse, dependiendo del resultado de la prueba `X .NE. 0.` En otras palabras, el valor de `Y` depende del flujo de control en el código a su alrededor. Nuevamente, esto puede sonarle a usted como un asunto que sólo le importa a los diseñadores de compiladores, y no a los programadores, cosa que es mayormente cierta. Pero hay veces que usted deseará mover tales instrucciones dependientes del control, para así quitar del camino costosos cálculos (suponiendo que su compilador no sea lo suficientemente inteligente para hacerlo por usted). Por ejemplo, digamos que [link] representa una pequeña sección de su programa. El flujo de control inicia en la parte superior y avanza a lo largo de dos decisiones. Por otro lado, digamos que hay una operación de raíz cuadrada en el punto de entrada, y que el flujo de control casi siempre inicia en la parte de arriba y baja por la rama que contiene la sentencia `A=0.0`. Ello significa que el resultado del cálculo `A=SQRT(B)` casi siempre se descargará, porque A obtiene un nuevo valor de 0.0 cada vez. Una operación de raíz cuadrada siempre es "cara", porque toma mucho tiempo en ejecutarse. El problema es que usted no puede simplemente quitarla de ahí, puesto que ocasionalmente se requiere. Sin embargo, puede quitarla del camino y continuar observando las dependencias e control, haciendo dos copias de la operación de raíz cuadrada a lo largo de la bifurcación menos recorrida, como se muestra en [link]. De esta forma el código `SQRT` sólo se ejecutará a lo largo de tales rutas cuando realmente se requiera.

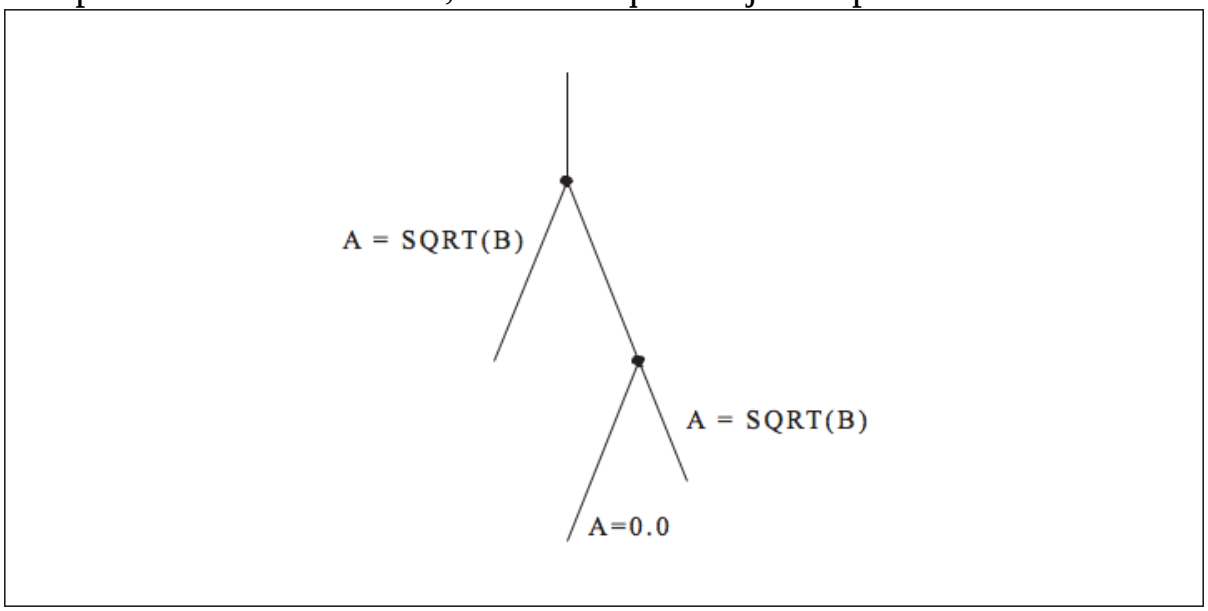
Dependencia de control



Una pequeña sección de nuestro programa



Esta clase de planificación de instrucciones irá apareciendo en los compiladores (e incluso en el hardware) más y más conforme pase el tiempo. Una variante de esta técnica es calcular los resultados que se requieran en aquellos momentos donde haya una brecha en el flujo de instrucciones (debido a las dependencias), usando así algunos ciclos sobrantes, que de otro modo se desperdiciarían.
La operación cara se movió, de forma que se ejecute pocas veces



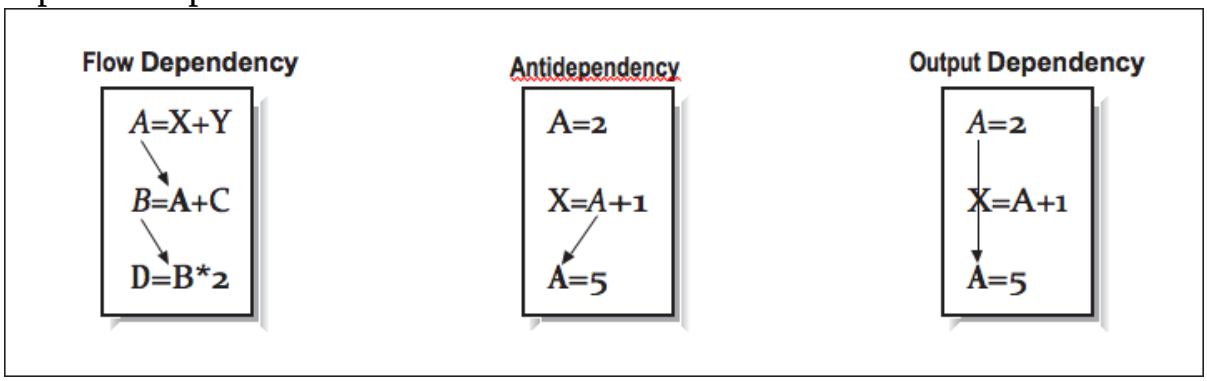
Dependencia de Datos

Un cálculo que está ligado en cierta forma con uno previo, se dice que presenta dependencia de los datos con respecto a este último. En el código siguiente, el valor de B es dependiente de los datos con respecto de A. Ello se debe a que no puede usted calcular B hasta que el valor de A esté disponible:

```
A = X + Y + COS(Z)
B = A * C
```

Esta dependencia es fácil de reorganizar, pero otras no lo son tanto. En otras ocasiones, deberá usted ser cuidadoso de no reescribir una variable con un nuevo valor antes de que otro cálculo haya terminado de usar el valor previo. Podemos agrupar todas las dependencias de datos en tres categorías: (1) dependencias del flujo, (2) antidependencias, y (3) dependencias de la salida. [link] contiene algunos ejemplos sencillos que demuestran cada tipo de dependencia. En cada ejemplo, usamos una flecha que inicia en el origen de la dependencia, y termina en la sentencia que debe retrasarse por causa de la dependencia. El problema clave en cada una de esas dependencias, es que la segunda sentencia no puede ejecutarse hasta que se haya completado la primera. Obviamente en el ejemplo de dependencia de salida en particular, el primer cálculo es código muerto y puede eliminarse, a menos que intervenga algún código que requiera de esos valores. Hay otras técnicas para eliminar ya sean las dependencias de salida o las antidependencias. El siguiente ejemplo contiene una dependencia de flujo seguida por una dependencia de salida:

Tipos de dependencias de datos



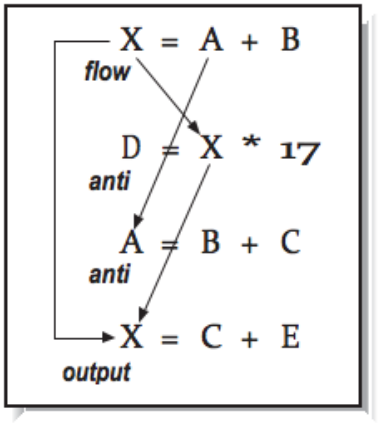
```
X = A / B
Y = X + 2.0
X = D - E
```

Aunque no podamos eliminar la dependencia de flujo, sí podemos lograrlo con la dependencia de salida agregando una variable nueva:

```
Xtemp = A/B
Y = Xtemp + 2.0
X = D - E
```

Conforme crece el número de sentencias e interacciones entre ellas, necesitamos una forma mejor de identificar y procesar tales dependencias. [link] muestra cuatro sentencias con cuatro dependencias.

Dependencias múltiples

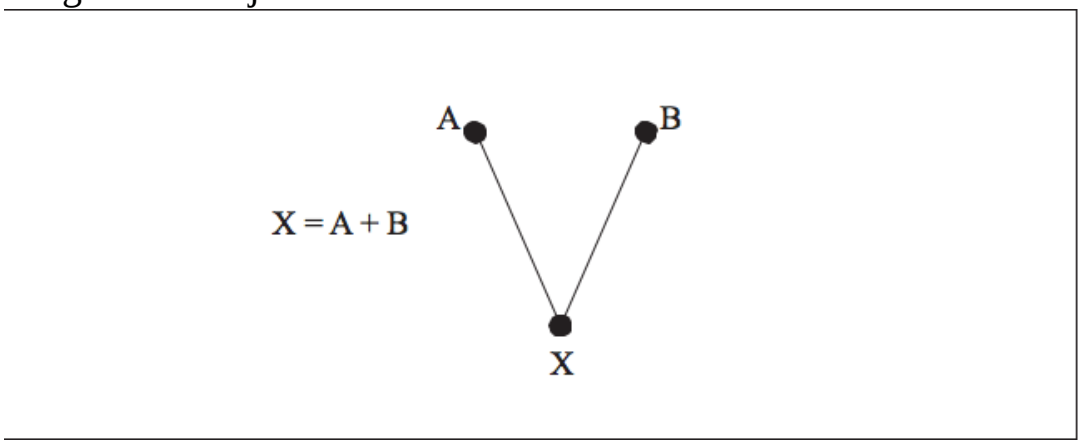


Ninguna de las instrucciones, de la segunda hasta la cuarta, pueden iniciarse antes de que se complete la primera instrucción.

Formando un GAD

Un método para analizar una secuencia de instrucciones, es organizarlas en un*grafo acíclico dirigido* (GAD).[footnote] Tal como las instrucciones que representa, un GAD describe todos los cálculos y relaciones entre variables. El flujo de datos en el GAD procede en una sola dirección; la mayoría de las ocasiones el GAD se construye de arriba hacia abajo. Los identificadores y las constantes se colocan en los nodos "hoja" -los que están en la parte superior. Las operaciones, posiblemente con los nombres de las variables adjuntos, se encuentran en los nodos internos. Las variables aparecen en sus estados finales en la parte inferior. Las aristas del GAD ordenan las relaciones entre las variables y las operaciones entre ellas. Todo el flujo de datos transcurre de arriba hacia abajo.
Un grafo es una colección de nodos conectados por aristas. Por dirigido nos referimos a que sólo pueden recorrerse las aristas en direcciones específicas. La palabra acíclico significa que no hay ciclos en el grafo, esto es, que no puede pasar dos veces por el mismo nodo.

Para construir un GAD, el compilador toma cada tupla de lenguaje intermedio y lo mapea sobre uno o más nodos. Por ejemplo, aquellas tuplas que representan operaciones binarias, tales como suma (`X=A+B`), forman una porción del GAD con dos entradas (`A` y `B`) enlazadas por una operación (`+`). El resultado de la operación puede alimentarse a su vez en otras operaciones en el bloque básico (y el GAD), como se muestra en [link].
Un grafo de flujo de datos trivial

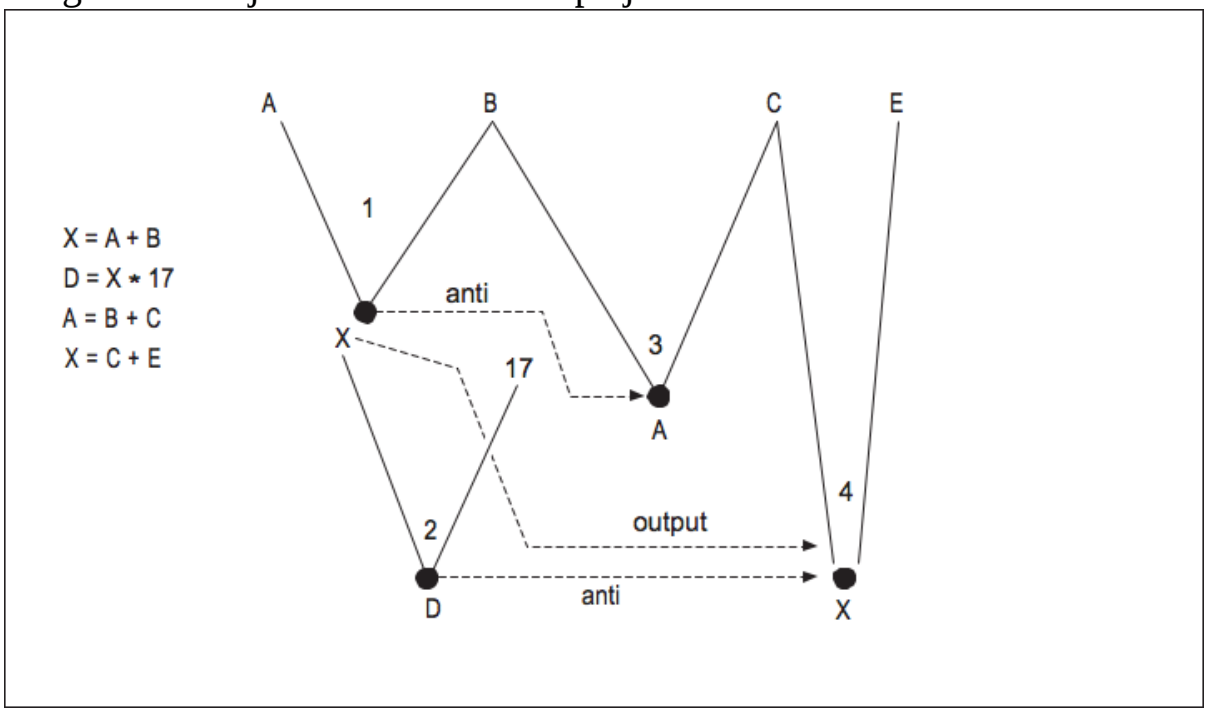


Si se trata de un bloque de código básico, construimos nuestro GAD en el orden de las instrucciones. El GAD de para las cuatro instrucciones previas

se muestra en [link]. Este ejemplo en particular tiene muchas dependencias, así que no hay mucha oportunidad para el paralelismo. [link] muestra un ejemplo más sencillo sobre cómo construir un GAD permite identificar el paralelismo.

A partir de este GAD, podemos determinar que las instrucciones 1 y 2 pueden ejecutarse en paralelo. Dado que vemos los cálculo que se realizan sobre los valores `A` y `B` durante el procesamiento de la instrucción 4, podemos eliminar una subexpresión común durante la construcción del GAD. Si podemos determinar que `Z` es la única variable que se usa afuera de este pequeño bloque de código, podemos asumir que el cálculo de `Y` es código muerto.

Un grafo de flujo de datos más complejo



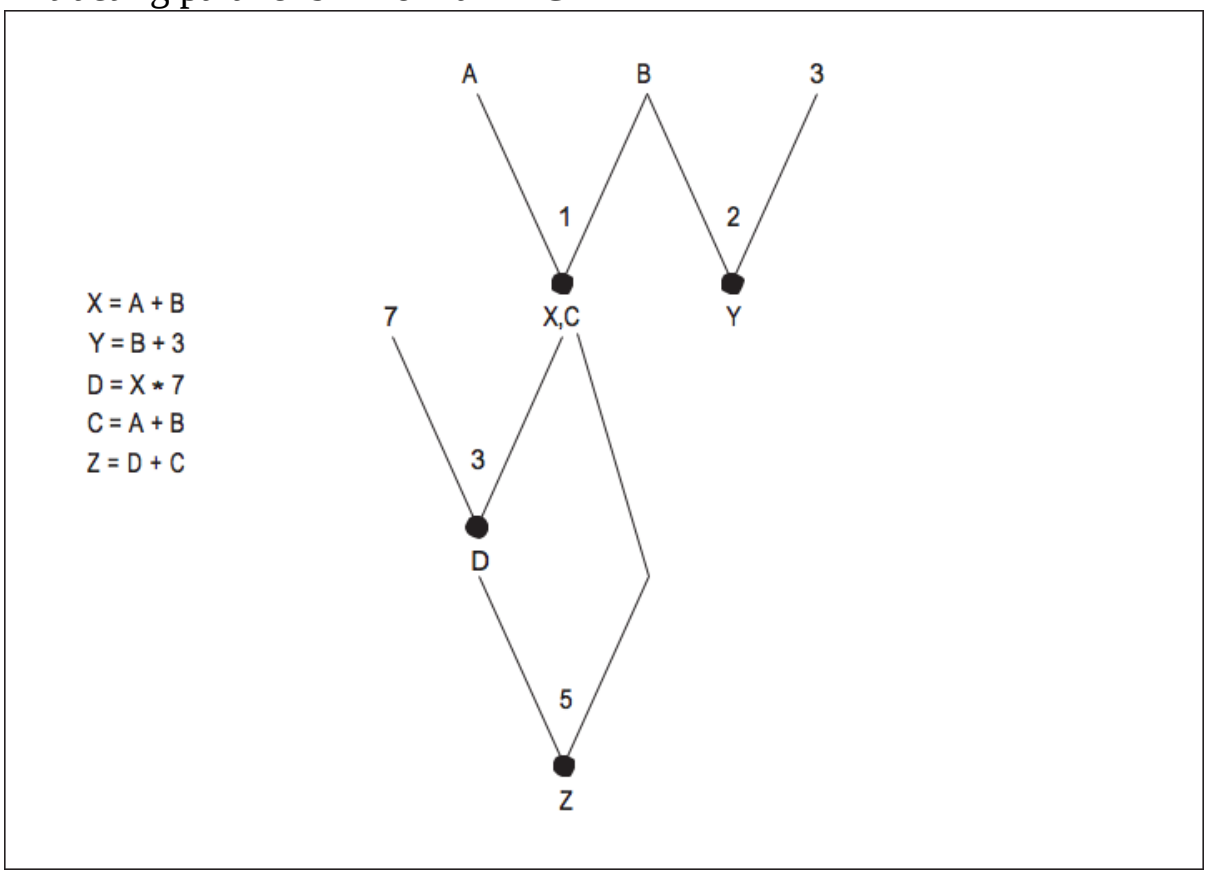
Al construir el GAD, tomamos una secuencia de instrucciones y determinamos cuáles deben ejecutarse en un orden particular, y cuáles pueden ejecutarse en paralelo. Este tipo de análisis de flujo de datos es muy importante en la fase de generación de código en los procesadores superescalares. Hemos introducido el concepto de dependencias y cómo usar el flujo de datos para encontrar oportunidades de paralelismo en secuencias de código adentro de un bloque básico. También podemos usar

el análisis de flujo de datos para identificar dependencias, oportunidades para el paralelismo, y código muerto al interior de bloques básicos.

Usos y Definiciones

Conforme se construye el GAD, el compilador puede crear listas de usos y *definiciones* de variables, así como otra información, y aplicarlas a las optimizaciones globales a través de muchos bloques básicos tomados junto. Al revisar el GAD en [link], podemos ver que las variables definidas son `Z`, `Y`, `X`, `C`, y `D`, y que las variables usadas son `A` y `B`. Al tomar en cuenta muchos bloques como uno solo, podemos decir cuan lejos alcanza la definición de una variable particular —dónde puede verse su valor. A partir de esto podemos reconocer situaciones en las cuales se descartan cálculos, lugares donde dos usos de una variable dada son completamente independientes, o dónde podemos reescribir valores residentes en registros sin tener que regresarlos a memoria. Llamamos a este tipo de investigación *análisis de flujo de datos*.

Extracting parallelism from a DAG



Para ilustrarlo, supongamos que tenemos el grafo de flujo de [link]. Junto a cada bloque básico hemos listado las variables que usa y las variables que define. ¿Qué nos puede decir al respecto el análisis del flujo de datos?

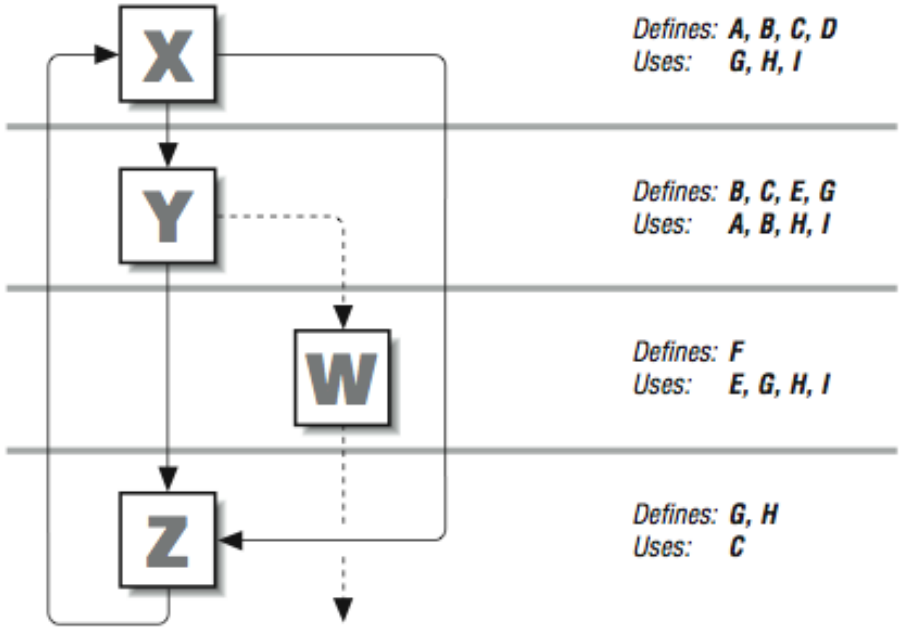
Observe que el valor de `A` se definió en el bloque `X`, pero sólo se usó en el bloque `Y`. Ello significa que `A` está muerto hasta la salida del bloque `Y` o inmediatamente antes de tomar la rama derecha al dejar `X`; ninguno de los otros bloques básicos usan el valor de A. Ello nos dice que cualesquiera recursos asociados, tales como un registro, pueden liberarse para otros usos.

En la [link] podemos ver que `D` está definida en el bloque básico `X`, pero jamás utilizada. Ello significa que los cálculos que definen a `D` pueden descartarse.

Algo interesante está sucediendo con la variable `G`. Tanto el bloque `X` como el `W` la utilizan, pero si observa detenidamente verá que los dos usos son distintos, significando que pueden tratarse como dos variables independientes.

Un compilador que instrumente técnicas avanzadas para la planificación adelantada de instrucciones, debe notar que `W` es el único bloque que usa el valor de `E`, y así mover los cálculos que definen `E` fuera del bloque `Y` y dentro de `W`, donde se requieren.

Grafo de flujo para análisis de flujo de datos.



Además de recopilar datos acerca de las variables, el compilador también puede mantener información acerca de subexpresiones. Al examinarlas juntas, puede reconocer casos donde se hacen cálculos redundantes (a lo largo de bloques básicos), y sustituirlos por cálculos previamente realizados. Si, por ejemplo, la expresión `H*I` aparece en los bloques `X`, `Y`, y `W`, puede calcularse sólo una vez en el bloque `X`, y propagarse a los otros que la usan.

Loops

Loops are the center of activity for many applications, so there is often a high payback for simplifying or moving calculations outside, into the computational suburbs. Early compilers for parallel architectures used pattern matching to identify the bounds of their loops. This limitation meant that a hand-constructed loop using if-statements and goto-statements would not be correctly identified as a loop. Because modern compilers use data flow graphs, it's practical to identify loops as a particular subset of nodes in the flow graph. To a data flow graph, a hand constructed loop looks the same as a compiler-generated loop. Optimizations can therefore be applied to either type of loop.

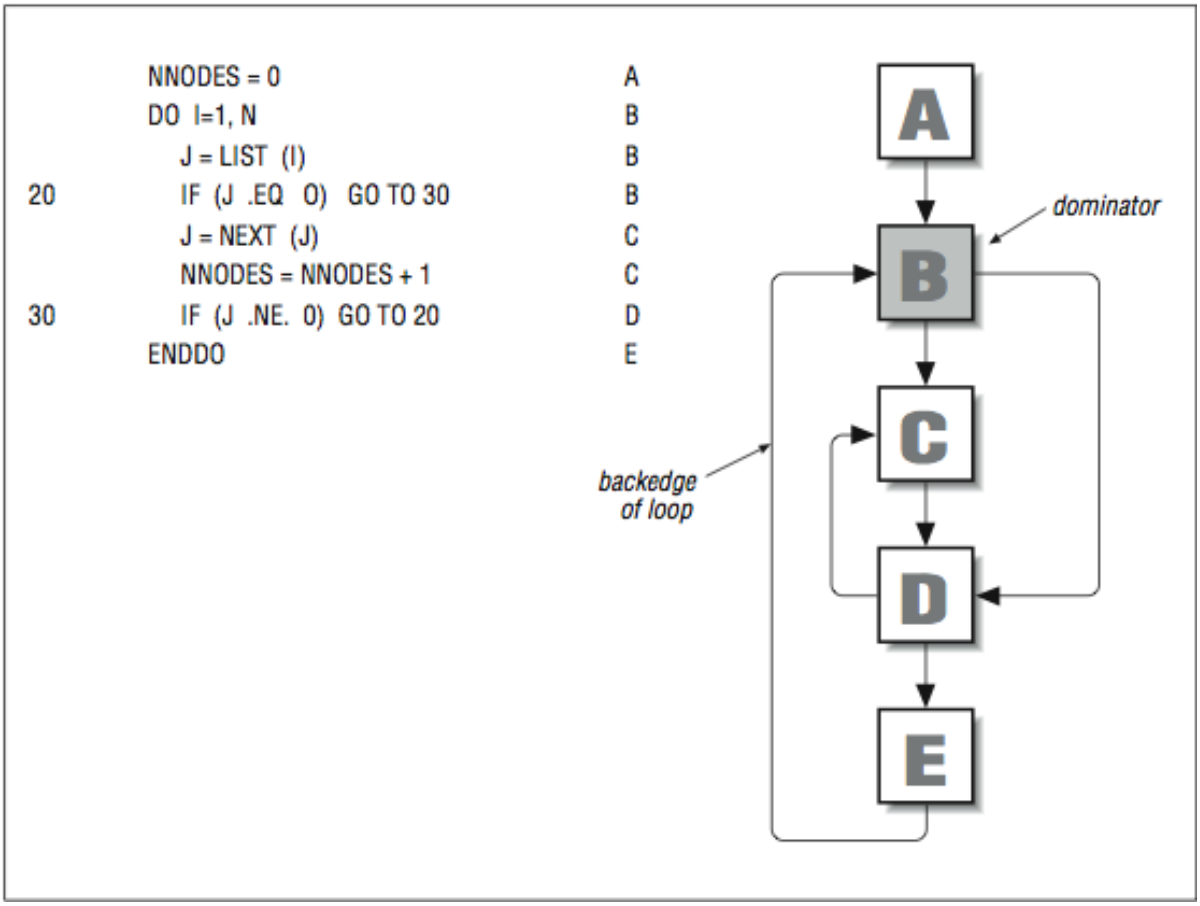
Once we have identified the loops, we can apply the same kinds of data-flow analysis we applied above. Among the things we are looking for are calculations that are unchanging within the loop and variables that change in a predictable (linear) fashion from iteration to iteration.

How does the compiler identify a loop in the flow graph? Fundamentally, two conditions have to be met:

- A given node has to dominate all other nodes within the suspected loop. This means that all paths to any node in the loop have to pass through one particular node, the dominator. The dominator node forms the header at the top of the loop.
- There has to be a cycle in the graph. Given a dominator, if we can find a path back to it from one of the nodes it dominates, we have a loop. This path back is known as the *back edge* of the loop.

The flow graph in [link] contains one loop and one red herring. You can see that node B dominates every node below it in the subset of the flow graph. That satisfies Condition 1 and makes it a candidate for a loop header. There is a path from `E` to `B`, and `B` dominates `E`, so that makes it a back edge, satisfying Condition 2. Therefore, the nodes `B`, `C`, `D`, and `E` form a loop. The loop goes through an array of linked list start pointers and traverses the lists to determine the total number of nodes in all lists. Letters to the extreme right correspond to the basic block numbers in the flow graph.

Flowgraph with a loop in it



At first glance, it appears that the nodes C and D form a loop too. The problem is that C doesn't dominate D (and vice versa), because entry to either can be made from B, so condition 1 isn't satisfied. Generally, the flow graphs that come from code segments written with even the weakest appreciation for a structured design offer better loop candidates.

After identifying a loop, the compiler can concentrate on that portion of the flow graph, looking for instructions to remove or push to the outside. Certain types of subexpressions, such as those found in array index expressions, can be simplified if they change in a predictable fashion from one iteration to the next.

In the continuing quest for parallelism, loops are generally our best sources for large amounts of parallelism. However, loops also provide new opportunities for those parallelism-killing dependencies.

Loop-Carried Dependencies

The notion of data dependence is particularly important when we look at loops, the hub of activity inside numerical applications. A well-designed loop can produce millions of operations that can all be performed in parallel. However, a single misplaced dependency in the loop can force it all to be run in serial. So the stakes are higher when looking for dependencies in loops.

Some constructs are completely independent, right out of the box. The question we want to ask is "Can two different iterations execute at the same time, or is there a data dependency between them?" Consider the following loop:

```
DO I=1,N
  A(I) = A(I) + B(I)
ENDDO
```

For any two values of `I` and `K`, can we calculate the value of `A(I)` and `A(K)` at the same time? Below, we have manually unrolled several iterations of the previous loop, so they can be executed together:

```
A(I) = A(I) + B(I)
A(I+1) = A(I+1) + B(I+1)
A(I+2) = A(I+2) + B(I+2)
```

You can see that none of the results are used as an operand for another calculation. For instance, the calculation for `A(I+1)` can occur at the same time as the calculation for `A(I)` because the calculations are independent; you don't need the results of the first to determine the second. In fact, mixing up the order of the calculations won't change the results in the least.

Relaxing the serial order imposed on these calculations makes it possible to execute this loop very quickly on parallel hardware.

Flow Dependencies

For comparison, look at the next code fragment:

```
DO I=2,N
  A(I) = A(I-1) + B(I)
ENDDO
```

This loop has the regularity of the previous example, but one of the subscripts is changed. Again, it's useful to manually unroll the loop and look at several iterations together:

```
A(I) = A(I-1) + B(I)
A(I+1) = A(I) + B(I+1)
A(I+2) = A(I+1) + B(I+2)
```

In this case, there is a dependency problem. The value of `A(I+1)` depends on the value of `A(I)`, the value of `A(I+2)` depends on `A(I+1)`, and so on; every iteration depends on the result of a previous one. Dependencies that extend back to a previous calculation and perhaps a previous iteration (like this one), are loop carried *flow dependencies* or *backward dependencies*. You often see such dependencies in applications that perform Gaussian elimination on certain types of matrices, or numerical solutions to systems of differential equations. However, it is impossible to run such a loop in parallel (as written); the processor must wait for intermediate results before it can proceed.

In some cases, flow dependencies are impossible to fix; calculations are so dependent upon one another that we have no choice but to wait for previous ones to complete. Other times, dependencies are a function of the way the calculations are expressed. For instance, the loop above can be changed to reduce the dependency. By replicating some of the arithmetic, we can make it so that the second and third iterations depend on the first, but not on one another. The operation count goes up — we have an extra addition that we didn't have before — but we have reduced the dependency between iterations:

```
DO I=2,N,2
  A(I)   = A(I-1) + B(I)
  A(I+1) = A(I-1) + B(I) + B(I+1)
ENDDO
```

The speed increase on a workstation won't be great (most machines run the recast loop more slowly). However, some parallel computers can trade off additional calculations for reduced dependency and chalk up a net win.

Antidependencies

It's a different story when there is a loop-carried antidependency, as in the code below:

```
DO I=1,N
  A(I)   = B(I)   * E
  B(I)   = A(I+2) * C
ENDDO
```

In this loop, there is an antidependency between the variable A(I) and the variable A(I+2). That is, you must be sure that the instruction that uses

A(I+2) does so before the previous one redefines it. Clearly, this is not a problem if the loop is executed serially, but remember, we are looking for opportunities to overlap instructions. Again, it helps to pull the loop apart and look at several iterations together. We have recast the loop by making many copies of the first statement, followed by copies of the second:

```
      A(I)   = B(I)   * E
      A(I+1) = B(I+1) * E
      A(I+2) = B(I+2) * E
      ...
      B(I)   = A(I+2) * C  ← assignment makes use
of the new
      B(I+1) = A(I+3) * C      value of A(I+2)
incorrect.
      B(I+2) = A(I+4) * C
```

The reference to `A(I+2)` needs to access an “old” value, rather than one of the new ones being calculated. If you perform all of the first statement followed by all of the second statement, the answers will be wrong. If you perform all of the second statement followed by all of the first statement, the answers will also be wrong. In a sense, to run the iterations in parallel, you must either save the `A` values to use for the second statement or store all of the `B` value in a temporary area until the loop completes.

We can also directly unroll the loop and find *some* parallelism:

```
      1  A(I)   = B(I)   * E
      2  B(I)   = A(I+2) * C →
      3  A(I+1) = B(I+1) * E  | Output dependency
      4  B(I+1) = A(I+3) * C  |
      5  A(I+2) = B(I+2) * E ←
```

```
6  B(I+2) = A(I+4) * C
```

Statements 1–4 could all be executed simultaneously. Once those statements completed execution, statements 5–8 could execute in parallel. Using this approach, there are sufficient intervening statements between the dependent statements that we can see some parallel performance improvement from a superscalar RISC processor.

Output Dependencies

The third class of data dependencies, *output dependencies*, is of particular interest to users of parallel computers, particularly multiprocessors. Output dependencies involve getting the right values to the right variables when all calculations have been completed. Otherwise, an output dependency is violated. The loop below assigns new values to two elements of the vector A with each iteration:

```
DO I=1,N
  A(I)   = C(I) * 2.
  A(I+2) = D(I) + E
ENDDO
```

As always, we won't have any problems if we execute the code sequentially. But if several iterations are performed together, and statements are reordered, then incorrect values can be assigned to the last elements of `A`. For example, in the naive vectorized equivalent below, `A(I+2)` takes the wrong value because the assignments occur out of order:

```
A(I)   = C(I)   * 2.
A(I+1) = C(I+1) * 2.
A(I+2) = C(I+2) * 2.
```

```
      A(I+2) = D(I)   + E ← Output dependency
violated
      A(I+3) = D(I+1) + E
      A(I+4) = D(I+2) + E
```

Whether or not you have to worry about output dependencies depends on whether you are actually parallelizing the code. Your compiler will be conscious of the danger, and will be able to generate legal code — and possibly even fast code, if it's clever enough. But output dependencies occasionally become a problem for programmers.

Dependencies Within an Iteration

We have looked at dependencies that cross iteration boundaries but we haven't looked at dependencies within the same iteration. Consider the following code fragment:

```
      DO I = 1,N
        D = B(I) * 17
        A(I) = D + 14
      ENDDO
```

When we look at the loop, the variable `D` has a flow dependency. The second statement cannot start until the first statement has completed. At first glance this might appear to limit parallelism significantly. When we look closer and manually unroll several iterations of the loop, the situation gets worse:

```
      D = B(I) * 17
      A(I) = D + 14
      D = B(I+1) * 17
```

```
A(I+1) = D + 14
D = B(I+2) * 17
A(I+2) = D + 14
```

Now, the variable `D` has flow, output, and antidependencies. It looks like this loop has no hope of running in parallel. However, there is a simple solution to this problem at the cost of some extra memory space, using a technique called *promoting a scalar to a vector*. We define `D` as an array `withN` elements and rewrite the code as follows:

```
DO I = 1,N
  D(I) = B(I) * 17
  A(I) = D(I) + 14
ENDDO
```

Now the iterations are all independent and can be run in parallel. Within each iteration, the first statement must run before the second statement.

Reductions

The sum of an array of numbers is one example of a *reduction* — so called because it reduces a vector to a scalar. The following loop to determine the total of the values in an array certainly looks as though it might be able to be run in parallel:

```
SUM = 0.0
DO I=1,N
  SUM = SUM + A(I)
ENDDO
```

However, if we perform our unrolling trick, it doesn’t look very parallel:

```
SUM = SUM + A(I)
SUM = SUM + A(I+1)
SUM = SUM + A(I+2)
```

This loop also has all three types of dependencies and looks impossible to parallelize. If we are willing to accept the potential effect of rounding, we can add some parallelism to this loop as follows (again we did not add the preconditioning loop):

```
SUM0 = 0.0
SUM1 = 0.0
SUM2 = 0.0
SUM3 = 0.0
DO I=1,N,4
  SUM0 = SUM0 + A(I)
  SUM1 = SUM1 + A(I+1)
  SUM2 = SUM2 + A(I+2)
  SUM3 = SUM3 + A(I+3)
ENDDO
SUM = SUM0 + SUM1 + SUM2 + SUM3
```

Again, this is not precisely the same computation, but all four partial sums can be computed independently. The partial sums are combined at the end of the loop.

Loops that look for the maximum or minimum elements in an array, or multiply all the elements of an array, are also reductions. Likewise, some of these can be reorganized into partial results, as with the sum, to expose more computations. Note that the maximum and minimum are associative

operators, so the results of the reorganized loop are identical to the sequential loop.

Ambiguous References

Every dependency we have looked at so far has been clear cut; you could see exactly what you were dealing with by looking at the source code. But other times, describing a dependency isn't so easy. Recall this loop from the "Antidependencies" section [link] earlier in this chapter:

```
DO I=1,N
  A(I) = B(I) * E
  B(I) = A(I+2) * C
ENDDO
```

Because each variable reference is solely a function of the index, `I`, it's clear what kind of dependency we are dealing with. Furthermore, we can describe how far apart (in iterations) a variable reference is from its definition. This is called the *dependency distance*. A negative value represents a flow dependency; a positive value means there is an antidependency. A value of zero says that no dependency exists between the reference and the definition. In this loop, the dependency distance for `A` is `+2` iterations.

However, array subscripts may be functions of other variables besides the loop index. It may be difficult to tell the distance between the use and definition of a particular element. It may even be impossible to tell whether the dependency is a flow dependency or an antidependency, or whether a dependency exists at all. Consequently, it may be impossible to determine if it's safe to overlap execution of different statements, as in the following loop:

```
DO I=1,N
  A(I) = B(I) * E
  B(I) = A(I+K) * C ← K unknown
```

```
ENDDO
```

If the loop made use of `A(I+K)`, where the value of `K` was unknown, we wouldn't be able to tell (at least by looking at the code) anything about the kind of dependency we might be facing. If `K` is zero, we have a dependency within the iteration and no loop-carried dependencies. If `K` is positive we have an antidependency with distance `K`. Depending on the value for `K`, we might have enough parallelism for a superscalar processor. If `K` is negative, we have a loop-carried flow dependency, and we may have to execute the loop serially.

Ambiguous references, like `A(I+K)` above, have an effect on the parallelism we can detect in a loop. From the compiler perspective, it may be that this loop does contain two independent calculations that the author whimsically decided to throw into a single loop. But when they appear together, the compiler has to treat them conservatively, as if they were interrelated. This has a big effect on performance. If the compiler has to assume that consecutive memory references may ultimately access the same location, the instructions involved cannot be overlapped. One other option is for the compiler to generate two versions of the loop and check the value for `K` at runtime to determine which version of the loop to execute.

A similar situation occurs when we use integer index arrays in a loop. The loop below contains only a single statement, but you can't be sure that any iteration is independent without knowing the contents of the `K` and `J` arrays:

```
DO I=1,N
  A(K(I)) = A(K(I)) + B(J(I)) * C
ENDDO
```

For instance, what if all of the values for `K(I)` were the same? This causes the same element of the array `A` to be rereferenced with each iteration! That may seem ridiculous to you, but the compiler can't tell.

With code like this, it's common for every value of `K(I)` to be unique. This is called a *permutation*. If you can tell a compiler that it is dealing with a permutation, the penalty is lessened in some cases. Even so, there is insult being added to injury. Indirect references require more memory activity than direct references, and this slows you down.

Pointer Ambiguity in Numerical C Applications

FORTRAN compilers depend on programmers to observe aliasing rules. That is, programmers are not supposed to modify locations through pointers that may be aliases of one another. They can become aliases in several ways, such as when two dummy arguments receive pointers to the same storage locations:

```
CALL BOB (A,A)
  ...
END
SUBROUTINE BOB (X,Y) ← X,Y become aliases
```

C compilers don't enjoy the same restrictions on aliasing. In fact, there are cases where aliasing could be desirable. Additionally, C is blessed with pointer types, increasing the opportunities for aliasing to occur. This means that a C compiler has to approach operations through pointers more conservatively than a FORTRAN compiler would. Let's look at some examples to see why.

The following loop nest looks like a FORTRAN loop cast in C. The arrays are declared or allocated all at once at the top of the routine, and the starting address and leading dimensions are visible to the compiler. This is important because it means that the storage relationship between the array elements is well known. Hence, you could expect good performance:

```
#define N ...
double *a[N][N], c[N][N], d;
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        a[i][j] = a[i][j] + c[j][i] * d;
```

Now imagine what happens if you allocate the rows dynamically. This makes the address calculations more complicated. The loop nest hasn't changed; however, there is no guaranteed stride that can get you from one row to the next. This is because the storage relationship between the rows is unknown:

```
#define N ...
double *a[N], *c[N], d;
    for (i=0; i<N; i++) {
        a[i] = (double *) malloc
(N*sizeof(double));
        c[i] = (double *) malloc
(N*sizeof(double));
    }
    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            a[i][j] = a[i][j] + c[j][i] * d;
```

In fact, your compiler knows even less than you might expect about the storage relationship. For instance, how can it be sure that references to a and c aren't aliases? It may be obvious to you that they're not. You might point out that *malloc* never overlaps storage. But the compiler isn't free to assume that. Who knows? You may be substituting your own version of *malloc*!

Let's look at a different example, where storage is allocated all at once, though the declarations are not visible to all routines that are using it. The following subroutine bob performs the same computation as our previous

example. However, because the compiler can’t see the declarations for a and c (they’re in the main routine), it doesn’t have enough information to be able to overlap memory references from successive iterations; the references could be aliases:

```
#define N...
main()
{
    double a[N][N], c[N][N], d;
    ...
    bob (a,c,d,N);
}
bob (double *a,double *c,double d,int n)
{
    int i,j;
    double *ap, *cp;
    for (i=0;i<n;i++) {
        ap = a + (i*n);
        cp = c + i;
        for (j=0; j<n; j++)
            *(ap+j) = *(ap+j) + *(cp+(j*n))
* d;
    }
}
```

To get the best performance, make available to the compiler as many details about the size and shape of your data structures as possible. Pointers, whether in the form of formal arguments to a subroutine or explicitly declared, can hide important facts about how you are using memory. The more information the compiler has, the more it can overlap memory references. This information can come from compiler directives or from making declarations visible in the routines where performance is most critical.

Closing Notes

You already knew there was a limit to the amount of parallelism in any given program. Now you know why. Clearly, if a program had no dependencies, you could execute the whole thing at once, given suitable hardware. But programs aren't infinitely parallel; they are often hardly parallel at all. This is because they contain dependencies of the types we saw above.

When we are writing and/or tuning our loops, we have a number of (sometimes conflicting) goals to keep in mind:

- Balance memory operations and computations.
- Minimize unnecessary operations.
- Access memory using unit stride if at all possible.
- Allow all of the loop iterations to be computed in parallel.

In the coming chapters, we will begin to learn more about executing our programs on parallel multiprocessors. At some point we will escape the bonds of compiler automatic optimization and begin to explicitly code the parallel portions of our code.

To learn more about compilers and dataflow, read The Art of Compiler Design: Theory and Practice by Thomas Pittman and James Peters (Prentice-Hall).

Exercises

Exercise:

Problem:

Identify the dependencies (if there are any) in the following loops. Can you think of ways to organize each loop for more parallelism?

a.
```
DO I=1,N-2 A(I+2) = A(I) + 1. ENDDO
```
b.
```
DO I=1,N-1,2 A(I+1) = A(I) + 1. ENDDO
```
c.
```
DO I=2,N A(I) = A(I-1) * 2. B = A(I-1) ENDDO
```
d.
```
DO I=1,N IF(N .GT. M) A(I) = 1. ENDDO
```
e.
```
DO I=1,N A(I,J) = A(I,K) + B ENDDO
```
f.
```
DO I=1,N-1 A(I+1,J) = A(I,K) + B ENDDO
```
g.
```
for (i=0; i<n; i++) a[i] = b[i];
```

Exercise:

Problem:

Imagine that you are a parallelizing compiler, trying to generate code for the loop below. Why are references to A a challenge? Why would it help to know that K is equal to zero? Explain how you could partially vectorize the statements involving A if you knew that K had an absolute value of at least 8.

```
DO I=1,N
  E(I,M) = E(I-1,M+1) - 1.0
  B(I) = A(I+K) * C
```

```
  A(I) = D(I) * 2.0
ENDDO
```

Exercise:

Problem:

The following three statements contain a flow dependency, an antidependency and an output dependency. Can you identify each? Given that you are allowed to reorder the statements, can you find a permutation that produces the same values for the variables `C` and `B`? Show how you can reduce the dependencies by combining or rearranging calculations and using temporary variables.

```
B = A + C
B = C + D
C = B + D
```

Introduction

In the mid-1980s, shared-memory multiprocessors were pretty expensive and pretty rare. Now, as hardware costs are dropping, they are becoming commonplace. Many home computer systems in the under-$3000 range have a socket for a second CPU. Home computer operating systems are providing the capability to use more than one processor to improve system performance. Rather than specialized resources locked away in a central computing facility, these shared-memory processors are often viewed as a logical extension of the desktop. These systems run the same operating system (UNIX or NT) as the desktop and many of the same applications from a workstation will execute on these multiprocessor servers.

Typically a workstation will have from 1 to 4 processors and a server system will have 4 to 64 processors. Shared-memory multiprocessors have a significant advantage over other multiprocessors because all the processors share the same view of the memory, as shown in [link].

These processors are also described as *uniform memory access* (also known as UMA) systems. This designation indicates that memory is equally accessible to all processors with the same performance.

The popularity of these systems is not due simply to the demand for high performance computing. These systems are excellent at providing high throughput for a multiprocessing load, and function effectively as high-performance database servers, network servers, and Internet servers. Within limits, their throughput is increased linearly as more processors are added.

In this book we are not so interested in the performance of database or Internet servers. That is too passé; buy more processors, get better throughput. We are interested in pure, raw, unadulterated compute speed for *our* high performance application. Instead of running hundreds of small jobs, we want to utilize all $750,000 worth of hardware for our single job.

The challenge is to find techniques that make a program that takes an hour to complete using one processor, complete in less than a minute using 64 processors. This is not trivial. Throughout this book so far, we have been on

an endless quest for parallelism. In this and the remaining chapters, we will begin to see the payoff for all of your hard work and dedication!

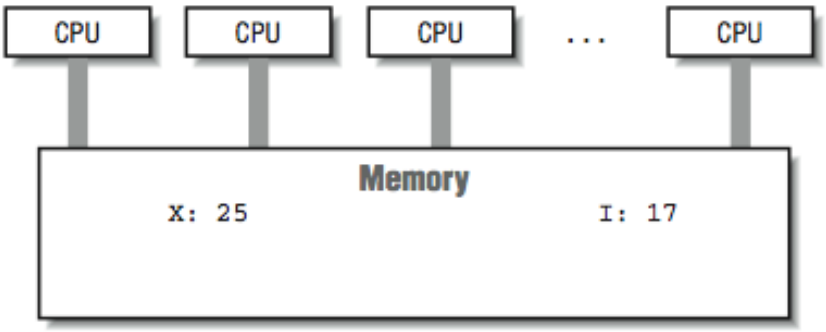
The cost of a shared-memory multiprocessor can range from $4000 to $30 million. Some example systems include multiple-processor Intel systems from a wide range of vendors, SGI Power Challenge Series, HP/Convex C-Series, DEC AlphaServers, Cray vector/parallel processors, and Sun Enterprise systems. The SGI Origin 2000, HP/Convex Exemplar, Data General AV-20000, and Sequent NUMAQ-2000 all are uniform-memory, symmetric multiprocessing systems that can be linked to form even larger shared nonuniform memory-access systems. Among these systems, as the price increases, the number of CPUs increases, the performance of individual CPUs increases, and the memory performance increases.

In this chapter we will study the hardware and software environment in these systems and learn how to execute our programs on these systems.

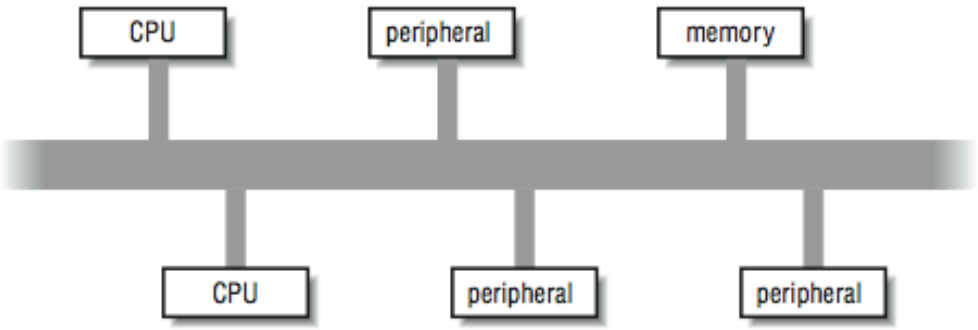
Symmetric Multiprocessing Hardware

In [link], we viewed an ideal shared-memory multiprocessor. In this section, we look in more detail at how such a system is actually constructed. The primary advantage of these systems is the ability for any CPU to access all of the memory and peripherals. Furthermore, the systems need a facility for deciding among themselves who has access to what, and when, which means there will have to be hardware support for arbitration. The two most common architectural underpinnings for symmetric multiprocessing are *buses* and *crossbars*. The bus is the simplest of the two approaches. [link] shows processors connected using a bus. A bus can be thought of as a set of parallel wires connecting the components of the computer (CPU, memory, and peripheral controllers), a set of protocols for communication, and some hardware to help carry it out. A bus is less expensive to build, but because all traffic must cross the bus, as the load increases, the bus eventually becomes a performance bottleneck.

A shared-memory multiprocessor



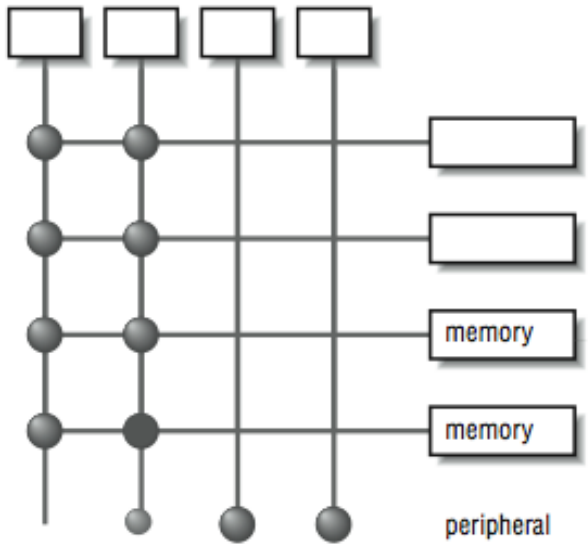
A typical bus architecture



A crossbar is a hardware approach to eliminate the bottleneck caused by a single bus. A crossbar is like several buses running side by side with attachments to each of the modules on the machine — CPU, memory, and peripherals. Any module can get to any other by a path through the crossbar, and multiple paths may be active simultaneously. In the 4×5 crossbar of [link], for instance, there can be four active data transfers in progress at one time. In the diagram it looks like a patchwork of wires, but there is actually quite a bit of hardware that goes into constructing a crossbar. Not only does the crossbar connect parties that wish to communicate, but it must also actively arbitrate between two or more CPUs that want access to the same memory or peripheral. In the event that one module is too popular, it's the crossbar that decides who gets access and who doesn't. Crossbars have the best performance because there is no single shared bus. However, they are more expensive to build, and their cost increases as the number of ports is increased. Because of their cost, crossbars typically are only found at the high end of the price and performance spectrum.

Whether the system uses a bus or crossbar, there is only so much memory bandwidth to go around; four or eight processors drawing from one memory system can quickly saturate all available bandwidth. All of the techniques that improve memory performance (as described in [link]) also apply here in the design of the memory subsystems attached to these buses or crossbars.

A crossbar



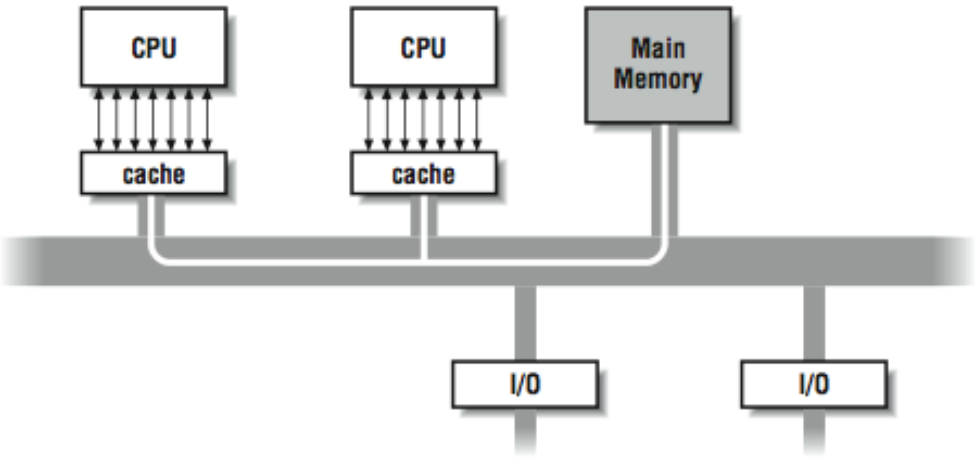
The Effect of Cache

The most common multiprocessing system is made up of commodity processors connected to memory and peripherals through a bus. Interestingly, the fact that these processors make use of cache somewhat mitigates the bandwidth bottleneck on a bus-based architecture. By connecting the processor to the cache and viewing the main memory through the cache, we significantly reduce the memory traffic across the bus. In this architecture, most of the memory accesses across the bus take the form of cache line loads and flushes. To understand why, consider what happens when the cache hit rate is very high. In [link], a high cache hit rate eliminates some of the traffic that would have otherwise gone out across the bus or crossbar to main memory. Again, it is the notion of “locality of reference” that makes the system work. If you assume that a fair number of the memory references will hit in the cache, the equivalent attainable main memory bandwidth is more than the bus is actually capable of. This assumption explains why multiprocessors are designed with less bus bandwidth than the sum of what the CPUs can consume at once.

Imagine a scenario where two CPUs are accessing different areas of memory using unit stride. Both CPUs access the first element in a cache line at the same time. The bus arbitrarily allows one CPU access to the

memory. The first CPU fills a cache line and begins to process the data. The instant the first CPU has completed its cache line fill, the cache line fill for the second CPU begins. Once the second cache line fill has completed, the second CPU begins to process the data in its cache line. If the time to process the data in a cache line is longer than the time to fill a cache line, the cache line fill for processor two completes before the next cache line request arrives from processor one. Once the initial conflict is resolved, both processors appear to have conflict-free access to memory for the remainder of their unit-stride loops.

High cache hit rate reduces main memory traffic



In actuality, on some of the fastest bus-based systems, the memory bus is sufficiently fast that up to 20 processors can access memory using unit stride with very little conflict. If the processors are accessing memory using non-unit stride, bus and memory bank conflict becomes apparent, with fewer processors.

This bus architecture combined with local caches is very popular for general-purpose multiprocessing loads. The memory reference patterns for database or Internet servers generally consist of a combination of time periods with a small working set, and time periods that access large data structures using unit stride. Scientific codes tend to perform more non-unit-stride access than general-purpose codes. For this reason, the most

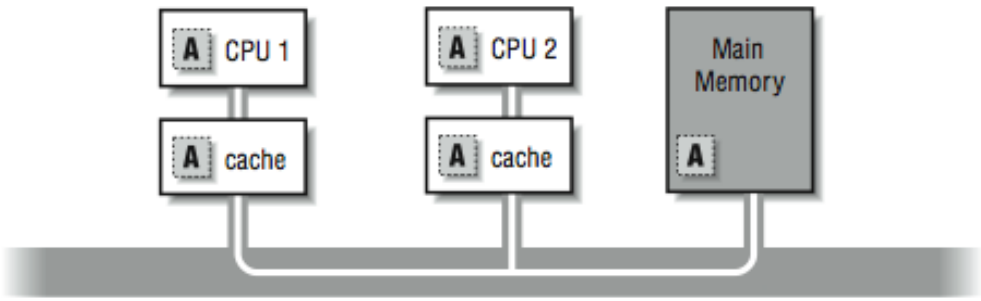
expensive parallel-processing systems targeted at scientific codes tend to use crossbars connected to multibanked memory systems.

The main memory system is better shielded when a larger cache is used. For this reason, multiprocessors sometimes incorporate a two-tier cache system, where each processor uses its own small on-chip local cache, backed up by a larger second board-level cache with as much as 4 MB of memory. Only when neither can satisfy a memory request, or when data has to be written back to main memory, does a request go out over the bus or crossbar.

Coherency

Now, what happens when one CPU of a multiprocessor running a single program in parallel changes the value of a variable, and another CPU tries to read it? Where does the value come from? These questions are interesting because there can be multiple copies of each variable, and some of them can hold old or stale values.

For illustration, say that you are running a program with a shared variable A. Processor 1 changes the value of A and Processor 2 goes to read it.
Multiple copies of variable A



In [link], if Processor 1 is keeping A as a register-resident variable, then Processor 2 doesn't stand a chance of getting the correct value when it goes to look for it. There is no way that 2 can know the contents of 1's registers; so assume, at the very least, that Processor 1 writes the new value back out. Now the question is, where does the new value get stored? Does it remain

in Processor 1's cache? Is it written to main memory? Does it get updated in Processor 2's cache?

Really, we are asking what kind of *cache coherency protocol* the vendor uses to assure that all processors see a uniform view of the values in "memory." It generally isn't something that the programmer has to worry about, except that in some cases, it can affect performance. The approaches used in these systems are similar to those used in single-processor systems with some extensions. The most straight-forward cache coherency approach is called a *write-through policy* : variables written into cache are simultaneously written into main memory. As the update takes place, other caches in the system see the main memory reference being performed. This can be done because all of the caches continuously monitor (also known as *snooping*) the traffic on the bus, checking to see if each address is in their cache. If a cache "notices" that it contains a copy of the data from the locations being written, it may either *invalidate* its copy of the variable or obtain new values (depending on the policy). One thing to note is that a write-through cache demands a fair amount of main memory bandwidth since each write goes out over the main memory bus. Furthermore, successive writes to the same location or bank are subject to the main memory cycle time and can slow the machine down.

A more sophisticated cache coherency protocol is called *copyback* or *writeback*. The idea is that you write values back out to main memory only when the cache housing them needs the space for something else. Updates of cached data are coordinated between the caches, by the caches, without help from the processor. Copyback caching also uses hardware that can monitor (snoop) and respond to the memory transactions of the other caches in the system. The benefit of this method over the write-through method is that memory traffic is reduced considerably. Let's walk through it to see how it works.

Cache Line States

For this approach to work, each cache must maintain a state for each line in its cache. The possible states used in the example include:

- **Modified**This cache line needs to be written back to memory.
- **Exclusive**There are no other caches that have this cache line.
- **Shared**There are read-only copies of this line in two or more caches.
- **Empty/Invalid**This cache line doesn’t contain any useful data.

This particular coherency protocol is often called *MESI*. Other cache coherency protocols are more complicated, but these states give you an idea how multiprocessor writeback cache coherency works.

We start where a particular cache line is in memory and in none of the writeback caches on the systems. The first cache to ask for data from a particular part of memory completes a normal memory access; the main memory system returns data from the requested location in response to a cache miss. The associated cache line is marked *exclusive*, meaning that this is the only cache in the system containing a copy of the data; it is the owner of the data. If another cache goes to main memory looking for the same thing, the request is intercepted by the first cache, and the data is returned from the first cache — not main memory. Once an interception has occurred and the data is returned, the data is marked *shared* in both of the caches.

When a particular line is marked shared, the caches have to treat it differently than they would if they were the exclusive owners of the data — especially if any of them wants to modify it. In particular, a write to a shared cache entry is preceded by a broadcast message to all the other caches in the system. It tells them to invalidate their copies of the data. The one remaining cache line gets marked as *modified* to signal that it has been changed, and that it must be returned to main memory when the space is needed for something else. By these mechanisms, you can maintain cache coherence across the multiprocessor without adding tremendously to the memory traffic.

By the way, even if a variable is not shared, it’s possible for copies of it to show up in several caches. On a symmetric multiprocessor, your program can bounce around from CPU to CPU. If you run for a little while on this CPU, and then a little while on that, your program will have operated out of separate caches. That means that there can be several copies of seemingly unshared variables scattered around the machine. Operating systems often try to minimize how often a process is moved between physical CPUs

during context switches. This is one reason not to overload the available processors in a system.

Data Placement

There is one more pitfall regarding shared memory we have so far failed to mention. It involves data movement. Although it would be convenient to think of the multiprocessor memory as one big pool, we have seen that it is actually a carefully crafted system of caches, coherency protocols, and main memory. The problems come when your application causes lots of data to be traded between the caches. Each reference that falls out of a given processor's cache (especially those that require an update in another processor's cache) has to go out on the bus.

Often, it's slower to get memory from another processor's cache than from the main memory because of the protocol and processing overhead involved. Not only do we need to have programs with high locality of reference and unit stride, we also need to minimize the data that must be moved from one CPU to another.

Multiprocessor Software Concepts

Now that we have examined the way shared-memory multiprocessor hardware operates, we need to examine how software operates on these types of computers. We still have to wait until the next chapters to begin making our FORTRAN programs run in parallel. For now, we use C programs to examine the fundamentals of multiprocessing and multithreading. There are several techniques used to implement multithreading, so the topics we will cover include:

- Operating system–supported multiprocessing
- User space multithreading
- Operating system-supported multithreading

The last of these is what we primarily will use to reduce the walltime of our applications.

Operating System–Supported Multiprocessing

Most modern general-purpose operating systems support some form of multiprocessing. Multiprocessing doesn't require more than one physical CPU; it is simply the operating system's ability to run more than one *process* on the system. The operating system context-switches between each process at fixed time intervals, or on interrupts or input-output activity. For example, in UNIX, if you use the `ps` command, you can see the processes on the system:

```
% ps -a
  PID TTY      TIME CMD
28410 pts/34   0:00 tcsh
28213 pts/38   0:00 xterm
10488 pts/51   0:01 telnet
28411 pts/34   0:00 xbiff
11123 pts/25   0:00 pine
 3805 pts/21   0:00 elm
```

```
 6773 pts/44  5:48 ansys
   ...
% ps –a | grep ansys
  6773 pts/44 6:00 ansys
```

For each process we see the process identifier (PID), the terminal that is executing the command, the amount of CPU time the command has used, and the name of the command. The PID is unique across the entire system. Most UNIX commands are executed in a separate process. In the above example, most of the processes are waiting for some type of event, so they are taking very few resources except for memory. Process 6773[footnote] seems to be executing and using resources. Running *ps* again confirms that the CPU time is increasing for the *ansys* process:
ANSYS is a commonly used structural-analysis package.

```
% vmstat 5
procs     memory          page            disk
faults     cpu
r b w   swap  free re mf pi po fr de sr f0 s0 -- -
-  in   sy  cs us sy id
3 0 0 353624 45432  0  0  1  0  0  0  0  0  0  0
0 461 5626 354 91  9  0
3 0 0 353248 43960  0 22  0  0  0  0  0  0 14  0
0 518 6227 385 89 11  0
```

Running the *vmstat 5* command tells us many things about the activity on the system. First, there are three runnable processes. If we had one CPU, only one would actually be running at a given instant. To allow all three jobs to progress, the operating system time-shares between the processes. Assuming equal priority, each process executes about 1/3 of the time. However, this system is a two-processor system, so each process executes about 2/3 of the time. Looking across the *vmstat* output, we can see paging activity (*pi*, *po*), context switches (*cs*), overall user time (*us*), system time (*sy*), and idle time (*id*).

Each process can execute a completely different program. While most processes are completely independent, they can cooperate and share information using interprocess communication (pipes, sockets) or various operating system-supported shared-memory areas. We generally don't use multiprocessing on these shared-memory systems as a technique to increase single-application performance.

Multiprocessing software

In this section, we explore how programs access multiprocessing features. [footnote] In this example, the program creates a new process using the `fork( )` function. The new process (child) prints some messages and then changes its identity using `exec( )` by loading a new program. The original process (parent) prints some messages and then waits for the child process to complete:
These examples are written in C using the POSIX 1003.1 application programming interface. This example runs on most UNIX systems and on other POSIX-compliant systems including OpenNT, Open- VMS, and many others.

```
int globvar;   /* A global variable */

main () {

 int pid,status,retval;
 int stackvar;   /* A stack variable */

 globvar = 1;
 stackvar = 1;
 printf("Main - calling fork globvar=%d
stackvar=%d\n",globvar,stackvar);
 pid = fork();
 printf("Main - fork returned pid=%d\n",pid);
 if ( pid == 0 ) {
   printf("Child - globvar=%d
```

```
stackvar=%d\n",globvar,stackvar);
   sleep(1);
   printf("Child - woke up globvar=%d
stackvar=%d\n",globvar,stackvar);
   globvar = 100;
   stackvar = 100;
   printf("Child - modified globvar=%d
stackvar=%d\n",globvar,stackvar);
   retval = execl("/bin/date", (char *) 0 );
   printf("Child - WHY ARE WE HERE
retval=%d\n",retval);
 } else {
   printf("Parent - globvar=%d
stackvar=%d\n",globvar,stackvar);
   globvar = 5;
   stackvar = 5;
   printf("Parent - sleeping globvar=%d
stackvar=%d\n",globvar,stackvar);
   sleep(2);
   printf("Parent - woke up globvar=%d
stackvar=%d\n",globvar,stackvar);
   printf("Parent - waiting for pid=%d\n",pid);
   retval = wait(&status);
   status = status >> 8; /* Return code in bits
15-8 */
   printf("Parent - status=%d
retval=%d\n",status,retval);
 }
}
```

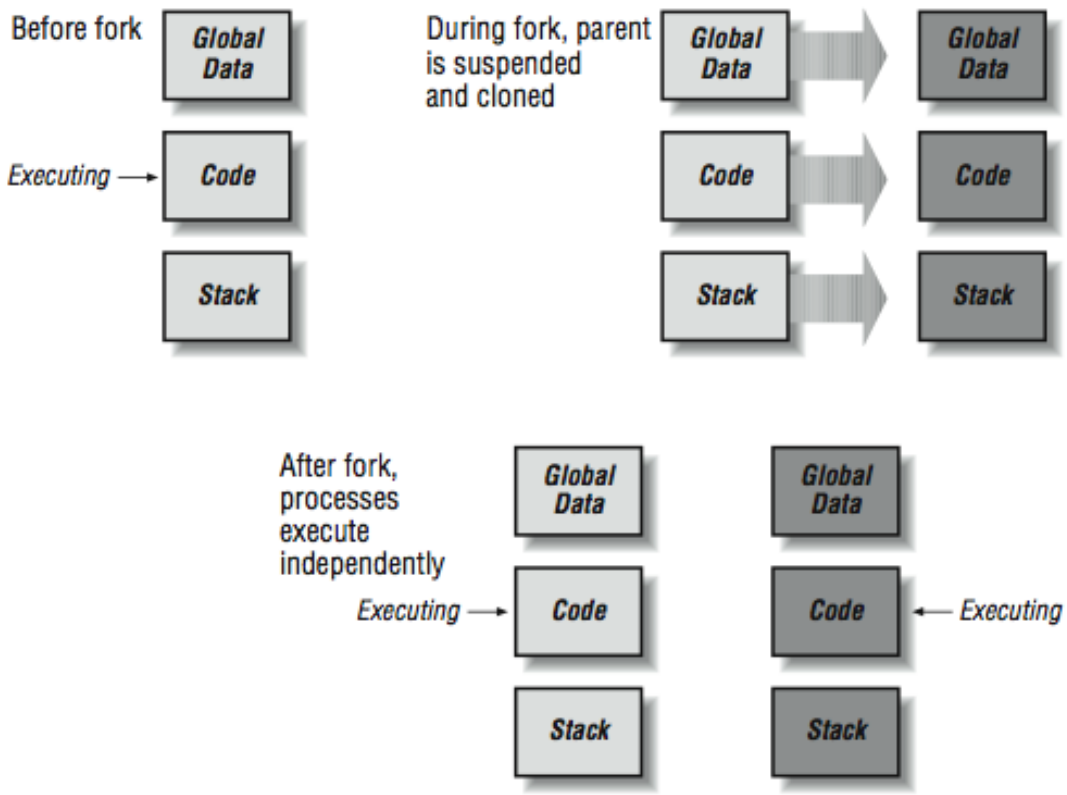
The key to understanding this code is to understand how the `fork( )` function operates. The simple summary is that the `fork( )` function is called once in a process and returns twice, once in the original process and once in a newly created process. The newly created process is an identical copy of the original process. All the variables (local and global) have been duplicated. Both processes have access to all of the open files of the original process. [link] shows how the fork operation creates a new process.

The only difference between the processes is that the return value from the `fork( )` function call is 0 in the new (child) process and the process identifier (shown by the `ps` command) in the original (parent) process. This is the program output:

```
recs % cc -o fork fork.c
recs % fork
Main - calling fork globvar=1 stackvar=1
Main - fork returned pid=19336
Main - fork returned pid=0
Parent - globvar=1 stackvar=1
Parent - sleeping globvar=5 stackvar=5
Child - globvar=1 stackvar=1
Child - woke up globvar=1 stackvar=1
Child - modified globvar=100 stackvar=100
Thu Nov 6 22:40:33
Parent - woke up globvar=5 stackvar=5
Parent - waiting for pid=19336
Parent - status=0 retval=19336
recs %
```

Tracing this through, first the program sets the global and stack variable to one and then calls `fork( )`. During the `fork( )` call, the operating system suspends the process, makes an exact duplicate of the process, and then restarts both processes. You can see two messages from the statement immediately after the fork. The first line is coming from the original process, and the second line is coming from the new process. If you were to execute a `ps` command at this moment in time, you would see two processes running called “fork.” One would have a process identifier of 19336.

How a fork operates



As both processes start, they execute an IF-THEN-ELSE and begin to perform different actions in the parent and child. Notice that *globvar* and *stackvar* are set to 5 in the parent, and then the parent sleeps for two seconds. At this point, the child begins executing. The values for *globvar* and *stackvar* are unchanged in the child process. This is because these two processes are operating in completely independent memory spaces. The child process sleeps for one second and sets its copies of the variables to 100. Next, the child process calls the `execl( )` function to overwrite its memory space with the UNIX date program. Note that the `execl( )` never returns; the date program takes over all of the resources of the child process. If you were to do a *ps* at this moment in time, you still see two processes on the system but process 19336 would be called "date." The date command executes, and you can see its output.[footnote]

It's not uncommon for a human parent process to "fork" and create a human child process that initially seems to have the same identity as the parent. It's also not uncommon for the child process to change its overall identity to be

something very different from the parent at some later point. Usually human children wait 13 years or so before this change occurs, but in UNIX, this happens in a few microseconds. So, in some ways, in UNIX, there are many parent processes that are “disappointed” because their children did not turn out like them!

The parent wakes up after a brief two-second sleep and notices that its copies of global and local variables have not been changed by the action of the child process. The parent then calls the `wait( )` function to determine if any of its children exited. The `wait( )` function returns which child has exited and the status code returned by that child process (in this case, process 19336).

User Space Multithreading

A *thread* is different from a process. When you add threads, they are added to the existing process rather than starting in a new process. Processes start with a single thread of execution and can add or remove threads throughout the duration of the program. Unlike processes, which operate in different memory spaces, all threads in a process share the same memory space. [link] shows how the creation of a thread differs from the creation of a process. Not all of the memory space in a process is shared between all threads. In addition to the global area that is shared across all threads, each thread has a *thread private* area for its own local variables. It’s important for programmers to know when they are working with shared variables and when they are working with local variables.

When attempting to speed up high performance computing applications, threads have the advantage over processes in that multiple threads can cooperate and work on a shared data structure to hasten the computation. By dividing the work into smaller portions and assigning each smaller portion to a separate thread, the total work can be completed more quickly.

Multiple threads are also used in high performance database and Internet servers to improve the overall *throughput* of the server. With a single thread, the program can either be waiting for the next network request or reading the disk to satisfy the previous request. With multiple threads, one

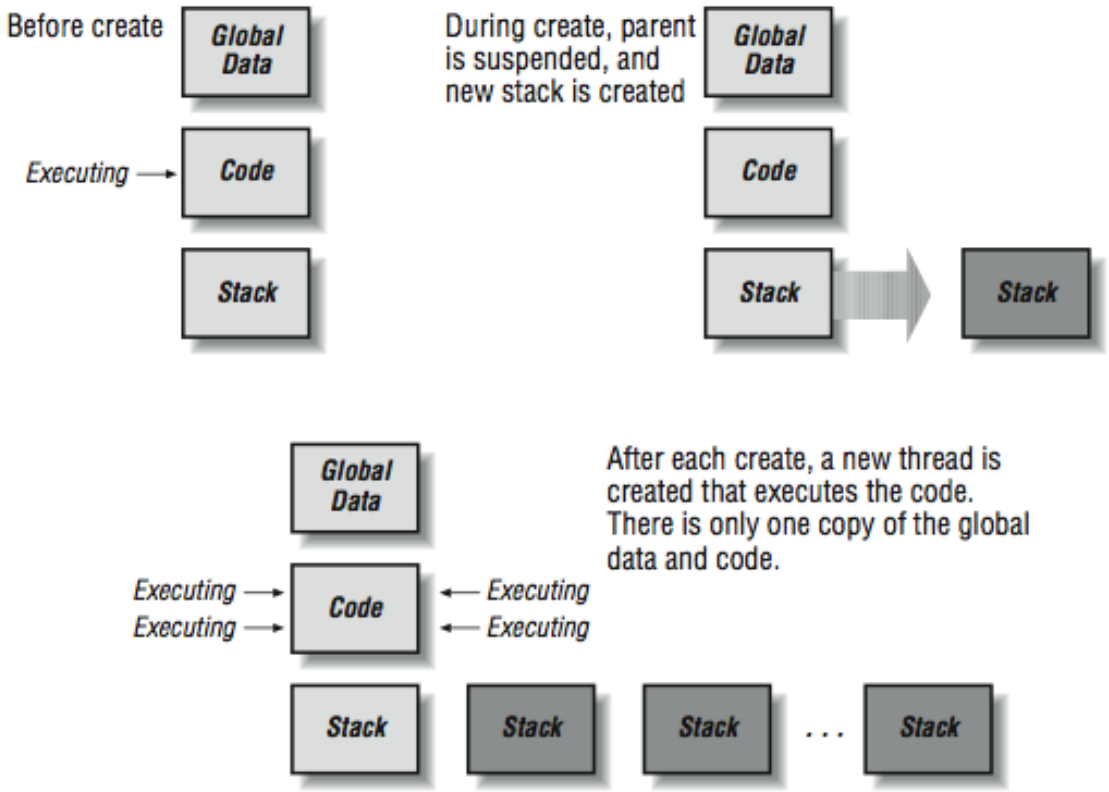
thread can be waiting for the next network transaction while several other threads are waiting for disk I/O to complete.

The following is an example of a simple multithreaded application. [footnote] It begins with a single master thread that creates three additional threads. Each thread prints some messages, accesses some global and local variables, and then terminates:
This example uses the IEEE POSIX standard interface for a thread library. If your system supports POSIX threads, this example should work. If not, there should be similar routines on your system for each of the thread functions.

```
#define_REENTRANT                    /* basic lines
for threads */
#include <stdio.h>
#include <pthread.h>

#define THREAD_COUNT 3
void *TestFunc(void *);
int globvar;                          /* A global
variable */
int index[THREAD_COUNT]               /* Local zero-
based thread index */
pthread_t thread_id[THREAD_COUNT];  /* POSIX
Thread IDs */

main() {
  int i,retval;
  pthread_t tid;

  globvar = 0;
  printf("Main - globvar=%d\n",globvar);
  for(i=0;i<THREAD_COUNT;i++) {
    index[i] = i;
    retval = pthread_create(&tid,NULL,TestFunc,
```

```
(void *) index[i]);
    printf("Main - creating i=%d tid=%d
retval=%d\n",i,tid,retval);
    thread_id[i] = tid;
  }
  printf("Main thread - threads started
globvar=%d\n",globvar);
  for(i=0;i<THREAD_COUNT;i++) {
    printf("Main - waiting for join
%d\n",thread_id[i]);
    retval = pthread_join( thread_id[i], NULL ) ;
    printf("Main - back from join %d
retval=%d\n",i,retval);
    }
    printf("Main thread - threads completed
globvar=%d\n",globvar);
  }

  void *TestFunc(void *parm) {
    int me,self;

    me = (int) parm; /* My own assigned thread
ordinal */
    self = pthread_self(); /* The POSIX Thread
library thread number */
    printf("TestFunc me=%d - self=%d
globvar=%d\n",me,self,globvar);
    globvar = me + 15;
    printf("TestFunc me=%d - sleeping
globvar=%d\n",me,globvar);
    sleep(2);
    printf("TestFunc me=%d - done param=%d
globvar=%d\n",me,self,globvar);
  }
```

Creating a thread



The global shared areas in this case are those variables declared in the static area outside the `main( )` code. The local variables are any variables declared within a routine. When threads are added, each thread gets its own function call stack. In C, the *automatic* variables that are declared at the beginning of each routine are allocated on the stack. As each thread enters a function, these variables are separately allocated on that particular thread's stack. So these are the *thread-local* variables.

Unlike the `fork( )` function, the `pthread_create( )` function creates a new thread, and then control is returned to the calling thread. One of the parameters of the `pthread_create( )` is the name of a function.

New threads begin execution in the function `TestFunc( )` and the thread finishes when it returns from this function. When this program is executed, it produces the following output:

```
    recs % cc -o create1 -lpthread -lposix4
create1.c
    recs % create1
    Main - globvar=0
    Main - creating i=0 tid=4 retval=0
    Main - creating i=1 tid=5 retval=0
    Main - creating i=2 tid=6 retval=0
    Main thread - threads started globvar=0
    Main - waiting for join 4
    TestFunc me=0 - self=4 globvar=0
    TestFunc me=0 - sleeping globvar=15
    TestFunc me=1 - self=5 globvar=15
    TestFunc me=1 - sleeping globvar=16
    TestFunc me=2 - self=6 globvar=16
    TestFunc me=2 - sleeping globvar=17
    TestFunc me=2 - done param=6 globvar=17
    TestFunc me=1 - done param=5 globvar=17
    TestFunc me=0 - done param=4 globvar=17
    Main - back from join 0 retval=0
    Main - waiting for join 5
    Main - back from join 1 retval=0
    Main - waiting for join 6
    Main - back from join 2 retval=0
    Main thread – threads completed globvar=17
    recs %
```

You can see the threads getting created in the loop. The master thread completes the `pthread_create( )` loop, executes the second loop, and calls the `pthread_join( )` function. This function suspends the master thread until the specified thread completes. The master thread is waiting for Thread 4 to complete. Once the master thread suspends, one of the new threads is started. Thread 4 starts executing. Initially the variable `globvar` is set to 0 from the main program. The `self, me`, and `param` variables are thread-local variables, so each thread has its own copy. Thread 4 sets

`globvar` to 15 and goes to sleep. Then Thread 5 begins to execute and sees `globvar` set to 15 from Thread 4; Thread 5 sets `globvar` to 16, and goes to sleep. This activates Thread 6, which sees the current value for `globvar` and sets it to 17. Then Threads 6, 5, and 4 wake up from their sleep, all notice the latest value of 17 in `globvar`, and return from the `TestFunc( )` routine, ending the threads.

All this time, the master thread is in the middle of `a pthread_join( )` waiting for Thread 4 to complete. As Thread 4 completes, the `pthread_join( )` returns. The master thread then calls `pthread_join( )` repeatedly to ensure that all three threads have been completed. Finally, the master thread prints out the value for `globvar` that contains the latest value of 17.

To summarize, when an application is executing with more than one thread, there are shared global areas and thread private areas. Different threads execute at different times, and they can easily work together in shared areas.

Limitations of user space multithreading

Multithreaded applications were around long before multiprocessors existed. It is quite practical to have multiple threads with a single CPU. As a matter of fact, the previous example would run on a system with any number of processors, including one. If you look closely at the code, it performs a sleep operation at each critical point in the code. One reason to add the sleep calls is to slow the program down enough that you can actually see what is going on. However, these sleep calls also have another effect. When one thread enters the sleep routine, it causes the thread library to search for other "runnable" threads. If a runnable thread is found, it begins executing immediately while the calling thread is "sleeping." This is called a *user-space thread context* switch. The process actually has one operating system thread shared among several logical user threads. When library routines (such as *sleep*) are called, the thread library[footnote] jumps in and reschedules threads.
The pthreads library supports both user-space threads and operating-system threads, as we shall soon see. Another popular early threads package was called *cthreads*.

We can explore this effect by substituting the following `SpinFunc( )` function, replacing `TestFunc( )` function in the `pthread_create( )` call in the previous example:

```
void *SpinFunc(void *parm) {
  int me;
  me = (int) parm;
  printf("SpinFunc me=%d - sleeping %d seconds
...\n", me, me+1);
  sleep(me+1);
  printf("SpinFunc me=%d – wake globvar=%d...\n",
me, globvar);
  globvar ++;
  printf("SpinFunc me=%d - spinning
globvar=%d...\n", me, globvar);
  while(globvar < THREAD_COUNT ) ;
  printf("SpinFunc me=%d – done globvar=%d...\n",
me, globvar);
  sleep(THREAD_COUNT+1);
}
```

If you look at the function, each thread entering this function prints a message and goes to sleep for 1, 2, and 3 seconds. Then the function increments `globvar` (initially set to 0 in main) and begins a while-loop, continuously checking the value of `globvar.` As time passes, the second and third threads should finish their `sleep( )`, increment the value for `globvar`, and begin the while-loop. When the last thread reaches the loop, the value for `globvar` is 3 and all the threads exit the loop. However, this isn't what happens:

```
recs % create2 &
[1] 23921
recs %
```

```
Main - globvar=0
Main - creating i=0 tid=4 retval=0
Main - creating i=1 tid=5 retval=0
Main - creating i=2 tid=6 retval=0
Main thread - threads started globvar=0
Main - waiting for join 4
SpinFunc me=0 - sleeping 1 seconds ...
SpinFunc me=1 - sleeping 2 seconds ...
SpinFunc me=2 - sleeping 3 seconds ...
SpinFunc me=0 - wake globvar=0...
SpinFunc me=0 - spinning globvar=1...

recs % ps
  PID TTY      TIME CMD
 23921 pts/35  0:09 create2
recs % ps
  PID TTY      TIME CMD
23921 pts/35   1:16 create2
recs % kill -9 23921
[1]    Killed                  create2
recs %
```

We run the program in the background[footnote] and everything seems to run fine. All the threads go to sleep for 1, 2, and 3 seconds. The first thread wakes up and starts the loop waiting for `globvar` to be incremented by the other threads. Unfortunately, with user space threads, there is no automatic time sharing. Because we are in a CPU loop that never makes a system call, the second and third threads never get scheduled so they can complete their `sleep( )` call. To fix this problem, we need to make the following change to the code:
Because we know it will hang and ignore interrupts.

```
while(globvar < THREAD_COUNT ) sleep(1) ;
```

With this sleep[footnote] call, Threads 2 and 3 get a chance to be "scheduled." They then finish their sleep calls, increment the `globvar` variable, and the program terminates properly.
Some thread libraries support a call to a routine sched_yield( ) that checks for runnable threads. If it finds a runnable thread, it runs the thread. If no thread is runnable, it returns immediately to the calling thread. This routine allows a thread that has the CPU to ensure that other threads make progress during CPU-intensive periods of its code.

You might ask the question, "Then what is the point of user space threads?" Well, when there is a high performance database server or Internet server, the multiple logical threads can overlap network I/O with database I/O and other background computations. This technique is not so useful when the threads all want to perform simultaneous CPU-intensive computations. To do this, you need threads that are created, managed, and scheduled by the operating system rather than a user library.

Operating System-Supported Multithreading

When the operating system supports multiple threads per process, you can begin to use these threads to do simultaneous computational activity. There is still no requirement that these applications be executed on a multiprocessor system. When an application that uses four operating system threads is executed on a single processor machine, the threads execute in a time-shared fashion. If there is no other load on the system, each thread gets 1/4 of the processor. While there are good reasons to have more threads than processors for noncompute applications, it's not a good idea to have more active threads than processors for compute-intensive applications because of thread-switching overhead. (For more detail on the effect of too many threads, see Appendix D, *How FORTRAN Manages Threads at Runtime*.

If you are using the POSIX threads library, it is a simple modification to request that your threads be created as operating-system rather rather than user threads, as the following code shows:

```
#define _REENTRANT      /* basic 3-lines for
threads */
#include <stdio.h>
#include <pthread.h>

#define THREAD_COUNT 2
void *SpinFunc(void *);
int globvar;                        /* A global
variable */
int index[THREAD_COUNT];            /* Local zero-
based thread index */
pthread_t thread_id[THREAD_COUNT];      /* POSIX
Thread IDs */
pthread_attr_t attr;                /* Thread
attributes NULL=use default */

main() {
  int i,retval;
  pthread_t tid;

  globvar = 0;
  pthread_attr_init(&attr);      /* Initialize
attr with defaults */
  pthread_attr_setscope(&attr,
PTHREAD_SCOPE_SYSTEM);
  printf("Main - globvar=%d\n",globvar);
  for(i=0;i<THREAD_COUNT;i++) {
    index[i] = i;
    retval = pthread_create(&tid,&attr,SpinFunc,
(void *) index[i]);
    printf("Main - creating i=%d tid=%d
retval=%d\n",i,tid,retval);
    thread_id[i] = tid;
  }
  printf("Main thread - threads started
```

```
globvar=%d\n",globvar);
  for(i=0;i<THREAD_COUNT;i++) {
    printf("Main - waiting for join
%d\n",thread_id[i]);
    retval = pthread_join( thread_id[i], NULL ) ;
    printf("Main - back from join %d
retval=%d\n",i,retval);
  }
  printf("Main thread - threads completed
globvar=%d\n",globvar);
}
```

The code executed by the master thread is modified slightly. We create an "attribute" data structure and set the `PTHREAD_SCOPE_SYSTEM` attribute to indicate that we would like our new threads to be created and scheduled by the operating system. We use the attribute information on the call to `pthread_create( )`. None of the other code has been changed. The following is the execution output of this new program:

```
recs % create3
Main - globvar=0
Main - creating i=0 tid=4 retval=0
SpinFunc me=0 - sleeping 1 seconds ...
Main - creating i=1 tid=5 retval=0
Main thread - threads started globvar=0
Main - waiting for join 4
SpinFunc me=1 - sleeping 2 seconds ...
SpinFunc me=0 - wake globvar=0...
SpinFunc me=0 - spinning globvar=1...
SpinFunc me=1 - wake globvar=1...
SpinFunc me=1 - spinning globvar=2...
SpinFunc me=1 - done globvar=2...
SpinFunc me=0 - done globvar=2...
Main - back from join 0 retval=0
Main - waiting for join 5
```

```
Main - back from join 1 retval=0
Main thread - threads completed globvar=2
recs %
```

Now the program executes properly. When the first thread starts spinning, the operating system is context switching between all three threads. As the threads come out of their `sleep( )`, they increment their shared variable, and when the final thread increments the shared variable, the other two threads instantly notice the new value (because of the cache coherency protocol) and finish the loop. If there are fewer than three CPUs, a thread may have to wait for a time-sharing context switch to occur before it notices the updated global variable.

With operating-system threads and multiple processors, a program can realistically break up a large computation between several independent threads and compute the solution more quickly. Of course this presupposes that the computation could be done in parallel in the first place.

Techniques for Multithreaded Programs

Given that we have multithreaded capabilities and multiprocessors, we must still convince the threads to work together to accomplish some overall goal. Often we need some ways to coordinate and cooperate between the threads. There are several important techniques that are used while the program is running with multiple threads, including:

- Fork-join (or create-join) programming
- Synchronization using a critical section with a lock, semaphore, or mutex
- Barriers

Each of these techniques has an overhead associated with it. Because these overheads are necessary to go parallel, we must make sure that we have sufficient work to make the benefit of parallel operation worth the cost.

Fork-Join Programming

This approach is the simplest method of coordinating your threads. As in the earlier examples in this chapter, a master thread sets up some global data structures that describe the tasks each thread is to perform and then use the `pthread_create( )` function to activate the proper number of threads. Each thread checks the global data structure using its thread-id as an index to find its task. The thread then performs the task and completes. The master thread waits at a `pthread_join( )` point, and when a thread has completed, it updates the global data structure and creates a new thread. These steps are repeated for each major iteration (such as a time-step) for the duration of the program:

```
      for(ts=0;ts<10000;ts++) { /* Time Step Loop
*/
         /* Setup tasks */
         for (ith=0;ith<NUM_THREADS;ith++)
pthread_create(..,work_routine,..)
```

```
         for (ith=0;ith<NUM_THREADS;ith++)
pthread_join(...)
      }
      work_routine() {
         /* Perform Task */
         return;
      }
```

The shortcoming of this approach is the overhead cost associated with creating and destroying an operating system thread for a potentially very short task.

The other approach is to have the threads created at the beginning of the program and to have them communicate amongst themselves throughout the duration of the application. To do this, they use such techniques as critical sections or barriers.

Synchronization

Synchronization is needed when there is a particular operation to a shared variable that can only be performed by one processor at a time. For example, in previous `SpinFunc( )` examples, consider the line:

```
globvar++;
```

In assembly language, this takes at least three instructions:

```
      LOAD   R1,globvar
      ADD    R1,1
      STORE  R1,globvar
```

What if `globvar` contained 0, Thread 1 was running, and, at the precise moment it completed the `LOAD` into Register R1 and before it had

completed the `ADD` or `STORE` instructions, the operating system interrupted the thread and switched to Thread 2? Thread 2 catches up and executes all three instructions using its registers: loading 0, adding 1 and storing the 1 back into `globvar`. Now Thread 2 goes to sleep and Thread 1 is restarted at the `ADD` instruction. Register R1 for Thread 1 contains the previously loaded value of 0; Thread 1 adds 1 and then stores 1 into `globvar`. What is wrong with this picture? We meant to use this code to count the number of threads that have passed this point. Two threads passed the point, but because of a bad case of bad timing, our variable indicates only that one thread passed. This is because the increment of a variable in memory is not `atomic`. That is, halfway through the increment, something else can happen.

Another way we can have a problem is on a multiprocessor when two processors execute these instructions simultaneously. They both do the `LOAD`, getting 0. Then they both add 1 and store 1 back to memory. [footnote] Which processor actually got the honor of storing *their* 1 back to memory is simply a race.
Boy, this is getting pretty picky. How often will either of these events really happen? Well, if it crashes your airline reservation system every 100,000 transactions or so, that would be way too often.

We must have some way of guaranteeing that only one thread can be in these three instructions at the same time. If one thread has started these instructions, all other threads must wait to enter until the first thread has exited. These areas are called *critical sections*. On single-CPU systems, there was a simple solution to critical sections: you could turn off interrupts for a few instructions and then turn them back on. This way you could guarantee that you would get all the way through before a timer or other interrupt occurred:

```
INTOFF                  // Turn off Interrupts
LOAD    R1,globvar
ADD     R1,1
STORE   R1,globvar
```

```
      INTON                    // Turn on Interrupts
```

However, this technique does not work for longer critical sections or when there is more than one CPU. In these cases, you need a lock, a semaphore, or a mutex. Most thread libraries provide this type of routine. To use a mutex, we have to make some modifications to our example code:

```
...
      pthread_mutex_t my_mutex; /* MUTEX data
structure */
        ...

      main() {
        ...
        pthread_attr_init(&attr); /* Initialize
attr with defaults */
        pthread_mutex_init (&my_mutex, NULL);
         .... pthread_create( ... )
          ...
      }
      void *SpinFunc(void *parm)    {
        ...
        pthread_mutex_lock (&my_mutex);
        globvar ++;
        pthread_mutex_unlock (&my_mutex);
        while(globvar < THREAD_COUNT ) ;
        printf("SpinFunc me=%d – done
globvar=%d...\n", me, globvar);
        ...
      }
```

The mutex data structure must be declared in the shared area of the program. Before the threads are created, `pthread_mutex_init` must be called to initialize the mutex. Before `globvar` is incremented, we must lock the mutex and after we finish updating `globvar` (three instructions later), we unlock the mutex. With the code as shown above, there will never

be more than one processor executing the `globvar++` line of code, and the code will never hang because an increment was missed. Semaphores and locks are used in a similar way.

Interestingly, when using user space threads, an attempt to lock an already locked mutex, semaphore, or lock can cause a thread context switch. This allows the thread that "owns" the lock a better chance to make progress toward the point where they will unlock the critical section. Also, the act of unlocking a mutex can cause the thread waiting for the mutex to be dispatched by the thread library.

Barriers

Barriers are different than critical sections. Sometimes in a multithreaded application, you need to have all threads arrive at a point before allowing any threads to execute beyond that point. An example of this is a *time-based simulation*. Each task processes its portion of the simulation but must wait until all of the threads have completed the current time step before any thread can begin the next time step. Typically threads are created, and then each thread executes a loop with one or more barriers in the loop. The rough pseudocode for this type of approach is as follows:

```
main() {
        for (ith=0;ith<NUM_THREADS;ith++)
pthread_create(..,work_routine,..)
        for (ith=0;ith<NUM_THREADS;ith++)
pthread_join(...) /* Wait a long time */
        exit()
      }

      work_routine() {

        for(ts=0;ts<10000;ts++) { /* Time Step
Loop */
               /* Compute total forces on
particles */
          wait_barrier();
```

```
                /* Update particle positions based
on the forces */
          wait_barrier();
        }
        return;
      }
```

In a sense, our `SpinFunc( )` function implements a barrier. It sets a variable initially to 0. Then as threads arrive, the variable is incremented in a critical section. Immediately after the critical section, the thread spins until the precise moment that all the threads are in the spin loop, at which time all threads exit the spin loop and continue on.

For a critical section, only one processor can be executing in the critical section at the same time. For a barrier, all processors must arrive at the barrier before any of the processors can leave.

A Real Example

In all of the above examples, we have focused on the mechanics of shared memory, thread creation, and thread termination. We have used the `sleep( )` routine to slow things down sufficiently to see interactions between processes. But we want to go very fast, not just learn threading for threading's sake.

The example code below uses the multithreading techniques described in this chapter to speed up a sum of a large array. The *hpcwall* routine is from [link].

This code allocates a four-million-element double-precision array and fills it with random numbers between 0 and 1. Then using one, two, three, and four threads, it sums up the elements in the array:

```
#define _REENTRANT        /* basic 3-lines for
threads */
#include <stdio.h>
#include <stdlib.h>

#include <pthread.h>

#define MAX_THREAD 4
void *SumFunc(void *);
int ThreadCount;                      /* Threads on
this try */
double GlobSum;                       /* A global
variable */
int index[MAX_THREAD];                /* Local zero-
based thread index */
pthread_t thread_id[MAX_THREAD];      /* POSIX
Thread IDs */
pthread_attr_t attr;                  /* Thread
attributes NULL=use default */
pthread_mutex_t my_mutex;             /* MUTEX data
structure */
```

```
structure */

#define MAX_SIZE 4000000
double array[MAX_SIZE];             /* What we are
summing... */
void hpcwall(double *);

main() {
  int i,retval;
  pthread_t tid;
  double single,multi,begtime,endtime;

  /* Initialize things */
  for (i=0; i<MAX_SIZE; i++) array[i] = drand48();
  pthread_attr_init(&attr);     /* Initialize attr
with defaults */
  pthread_mutex_init (&my_mutex, NULL);
  pthread_attr_setscope(&attr,
PTHREAD_SCOPE_SYSTEM);

  /* Single threaded sum */
  GlobSum = 0;

  hpcwall(&begtime);
  for(i=0; i<MAX_SIZE;i++) GlobSum = GlobSum +
array[i];
  hpcwall(&endtime);
  single = endtime - begtime;
  printf("Single sum=%lf
time=%lf\n",GlobSum,single);

  /* Use different numbers of threads to
accomplish the same thing */
  for(ThreadCount=2;ThreadCount<=MAX_THREAD;
ThreadCount++) {
    printf("Threads=%d\n",ThreadCount);
    GlobSum = 0;
    hpcwall(&begtime);
    for(i=0;i<ThreadCount;i++) {
```

```
    for(i=0;i<ThreadCount;i++) {
      index[i] = i;
      retval = pthread_create(&tid,&attr,SumFunc,
(void *) index[i]);
      thread_id[i] = tid;
    }
    for(i=0;i<ThreadCount;i++) retval =
pthread_join(thread_id[i],NULL);
    hpcwall(&endtime);
    multi = endtime - begtime;
    printf("Sum=%lf time=%lf\n",GlobSum,multi);
    printf("Efficiency =
%lf\n",single/(multi*ThreadCount));
  } /* End of the ThreadCount loop */
}

void *SumFunc(void *parm){
  int i,me,chunk,start,end;
  double LocSum;

  /* Decide which iterations belong to me */
  me = (int) parm;

  chunk = MAX_SIZE / ThreadCount;
  start = me * chunk;
  end = start + chunk; /* C-Style - actual element
+ 1 */
  if ( me == (ThreadCount-1) ) end = MAX_SIZE;
  printf("SumFunc me=%d start=%d
end=%d\n",me,start,end);

  /* Compute sum of our subset*/
  LocSum = 0;
  for(i=start;i<end;i++ ) LocSum = LocSum +
array[i];

  /* Update the global sum and return to the
waiting join */
  pthread_mutex_lock (&my_mutex);
```

```
  pthread_mutex_lock (&my_mutex);
  GlobSum = GlobSum + LocSum;
  pthread_mutex_unlock (&my_mutex);
}
```

First, the code performs the sum using a single thread using a for-loop. Then for each of the parallel sums, it creates the appropriate number of threads that call `SumFunc( )`. Each thread starts in `SumFunc( )` and initially chooses an area to operation in the shared array. The “strip” is chosen by dividing the overall array up evenly among the threads with the last thread getting a few extra if the division has a remainder.

Then, each thread independently performs the sum on its area. When a thread has finished its computation, it uses a mutex to update the global sum variable with its contribution to the global sum:

```
recs % addup
Single sum=7999998000000.000000 time=0.256624
Threads=2
SumFunc me=0 start=0 end=2000000
SumFunc me=1 start=2000000 end=4000000
Sum=7999998000000.000000 time=0.133530
Efficiency = 0.960923
Threads=3
SumFunc me=0 start=0 end=1333333
SumFunc me=1 start=1333333 end=2666666
SumFunc me=2 start=2666666 end=4000000
Sum=7999998000000.000000 time=0.091018
Efficiency = 0.939829
Threads=4
SumFunc me=0 start=0 end=1000000
SumFunc me=1 start=1000000 end=2000000
SumFunc me=2 start=2000000 end=3000000
SumFunc me=3 start=3000000 end=4000000
Sum=7999998000000.000000 time=0.107473
```

```
Efficiency = 0.596950
recs %
```

There are some interesting patterns. Before you interpret the patterns, you must know that this system is a three-processor Sun Enterprise 3000. Note that as we go from one to two threads, the time is reduced to one-half. That is a good result given how much it costs for that extra CPU. We characterize how well the additional resources have been used by computing an efficiency factor that should be 1.0. This is computed by multiplying the wall time by the number of threads. Then the time it takes on a single processor is divided by this number. If you are using the extra processors well, this evaluates to 1.0. If the extra processors are used pretty well, this would be about 0.9. If you had two threads, and the computation did not speed up at all, you would get 0.5.

At two and three threads, wall time is dropping, and the efficiency is well over 0.9. However, at four threads, the wall time increases, and our efficiency drops very dramatically. This is because we now have more threads than processors. Even though we have four threads that could execute, they must be time-sliced between three processors.[footnote] This is even worse that it might seem. As threads are switched, they move from processor to processor and their caches must also move from processor to processor, further slowing performance. This cache-thrashing effect is not too apparent in this example because the data structure is so large, most memory references are not to values previously in cache.
It is important to match the number of runnable threads to the available resources. In compute code, when there are more threads than available processors, the threads compete among themselves, causing unnecessary overhead and reducing the efficiency of your computation.

It's important to note that because of the nature of floating-point (see [link]), the parallel sum may not be the same as the serial sum. To perform a summation in parallel, you must be willing to tolerate these slight variations in your results.

Closing Notes

As they drop in price, multiprocessor systems are becoming far more common. These systems have many attractive features, including good price/performance, compatibility with workstations, large memories, high throughput, large shared memories, fast I/O, and many others. While these systems are strong in multiprogrammed server roles, they are also an affordable high performance computing resource for many organizations. Their cache-coherent shared-memory model allows multithreaded applications to be easily developed.

We have also examined some of the software paradigms that must be used to develop multithreaded applications. While you hopefully will never have to write C code with explicit threads like the examples in this chapter, it is nice to understand the fundamental operations at work on these multiprocessor systems. Using the FORTRAN language with an automatic parallelizing compiler, we have the advantage that these and many more details are left to the FORTRAN compiler and runtime library. At some point, especially on the most advanced architectures, you may have to explicitly program a multithreaded program using the types of techniques shown in this chapter.

One trend that has been predicted for some time is that we will begin to see multiple cache-coherent CPUs on a single chip once the ability to increase the clock rate on a single chip slows down. Imagine that your new $2000 workstation has four 1-GHz processors on a single chip. Sounds like a good time to learn how to write multithreaded programs!

Exercises

Exercise:

Problem:

Experiment with the fork code in this chapter. Run the program multiple times and see how the order of the messages changes. Explain the results.

Exercise:

Problem:

Experiment with the `create1` and `create3` codes in this chapter. Remove all of the `sleep( )` calls. Execute the programs several times on single and multiprocessor systems. Can you explain why the output changes from run to run in some situations and doesn't change in others?

Exercise:

Problem:

Experiment with the parallel sum code in this chapter. In the `SumFunc( )` routine, change the for-loop to:

```
for(i=start;i<end;i++ ) GlobSum = GlobSum +
array[i];
```

Remove the three lines at the end that get the mutex and update the `GlobSum`. Execute the code. Explain the difference in values that you see for `GlobSum`. Are the patterns different on a single processor and a multiprocessor? Explain the performance impact on a single processor and a multiprocessor.

Exercise:

Problem:

Explain how the following code segment could cause deadlock — two or more processes waiting for a resource that can't be relinquished:

```
...
    call lock (lword1)
    call lock (lword2)
    ...
    call unlock (lword1)
    call unlock (lword2)
    .
    .
    .
    call lock (lword2)
    call lock (lword1)
       ...
    call unlock (lword2)
    call unlock (lword1)
       ...
```

Exercise:

Problem:

If you were to code the functionality of a spin-lock in C, it might look like this:

```
while (!lockword);
lockword = !lockword;
```

As you know from the first sections of the book, the same statements would be compiled into explicit loads and stores, a comparison, and a branch. There's a danger that two processes could each load `lockword`, find it unset, and continue on as if they owned the lock (we have a race condition). This suggests that spin-locks are implemented differently — that they're not merely the two lines of C above. How do you suppose they are implemented?

Introduction

In [link], we examined the hardware used to implement shared-memory parallel processors and the software environment for a programmer who is using threads explicitly. In this chapter, we view these processors from a simpler vantage point. When programming these systems in FORTRAN, you have the advantage of the compiler's support of these systems. At the top end of ease of use, we can simply add a flag or two on the compilation of our well-written code, set an environment variable, and voilá, we are executing in parallel. If you want some more control, you can add directives to particular loops where you know better than the compiler how the loop should be executed.[footnote] First we examine how well-written loops can benefit from automatic parallelism. Then we will look at the types of directives you can add to your program to assist the compiler in generating parallel code. While this chapter refers to running your code in parallel, most of the techniques apply to the vector-processor supercomputers as well.

If you have skipped all the other chapters in the book and jumped to this one, don't be surprised if some of the terminology is unfamiliar. While all those chapters seemed to contain endless boring detail, they did contain some basic terminology. So those of us who read all those chapters have some common terminology needed for this chapter. If you don't go back and read all the chapters, don't complain about the big words we keep using in this chapter!

Automatic Parallelization

So far in the book, we've covered the tough things you need to know to do parallel processing. At this point, assuming that your loops are clean, they use unit stride, and the iterations can all be done in parallel, all you have to do is turn on a compiler flag and buy a good parallel processor. For example, look at the following code:

```
      PARAMETER(NITER=300,N=1000000)
      REAL*8 A(N),X(N),B(N),C
      DO ITIME=1,NITER
        DO I=1,N
            A(I) = X(I) + B(I) * C
        ENDDO
        CALL WHATEVER(A,X,B,C)
      ENDDO
```

Here we have an iterative code that satisfies all the criteria for a good parallel loop. On a good parallel processor with a modern compiler, you are two flags away from executing in parallel. On Sun Solaris systems, the `autopar` flag turns on the automatic parallelization, and the `loopinfo` flag causes the compiler to describe the particular optimization performed for each loop. To compile this code under Solaris, you simply add these flags to your `f77` call:

```
      E6000: f77 -O3 -autopar -loopinfo -o daxpy
daxpy.f
      daxpy.f:
      "daxpy.f", line 6: not parallelized, call
may be unsafe
      "daxpy.f", line 8: PARALLELIZED
      E6000: /bin/time daxpy
```

```
real        30.9
user        30.7
sys          0.1
E6000:
```

If you simply run the code, it's executed using one thread. However, the code is enabled for parallel processing for those loops that can be executed in parallel. To execute the code in parallel, you need to set the UNIX environment to the number of parallel threads you wish to use to execute the code. On Solaris, this is done using the `PARALLEL` variable:

```
E6000: setenv PARALLEL 1
E6000: /bin/time daxpy

real         30.9
user         30.7
sys           0.1
E6000: setenv PARALLEL 2
E6000: /bin/time daxpy

real         15.6
user         31.0
sys           0.2
E6000: setenv PARALLEL 4
E6000: /bin/time daxpy

real          8.2
user         32.0
sys           0.5
E6000: setenv PARALLEL 8
E6000: /bin/time daxpy

real          4.3
user         33.0
```

```
sys          0.8
```

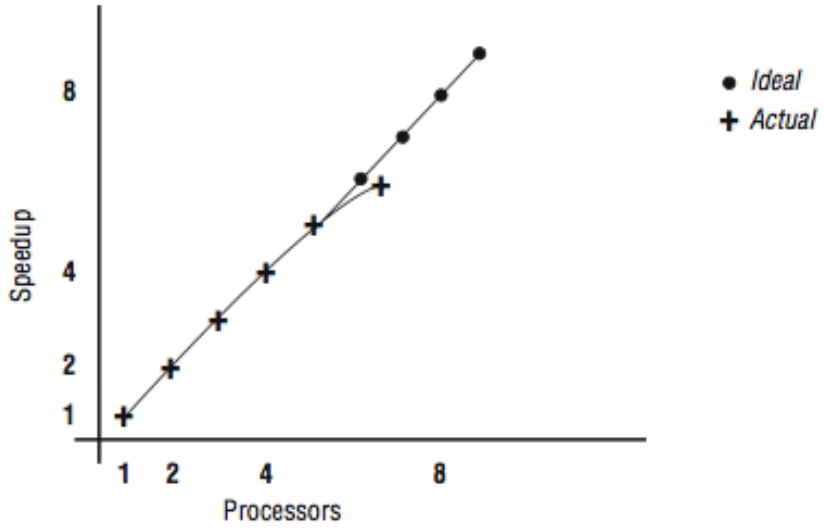
Speedup is the term used to capture how much faster the job runs using N processors compared to the performance on one processor. It is computed by dividing the single processor time by the multiprocessor time for each number of processors. [link] shows the wall time and speedup for this application.

Improving performance by adding processors

Processors	Time	Speedup
1	30.9	1.0 *(by definition)*
2	15.6	1.98
4	8.2	3.77
8	4.3	7.19

[link] shows this information graphically, plotting speedup versus the number of processors.

Ideal and actual performance improvement



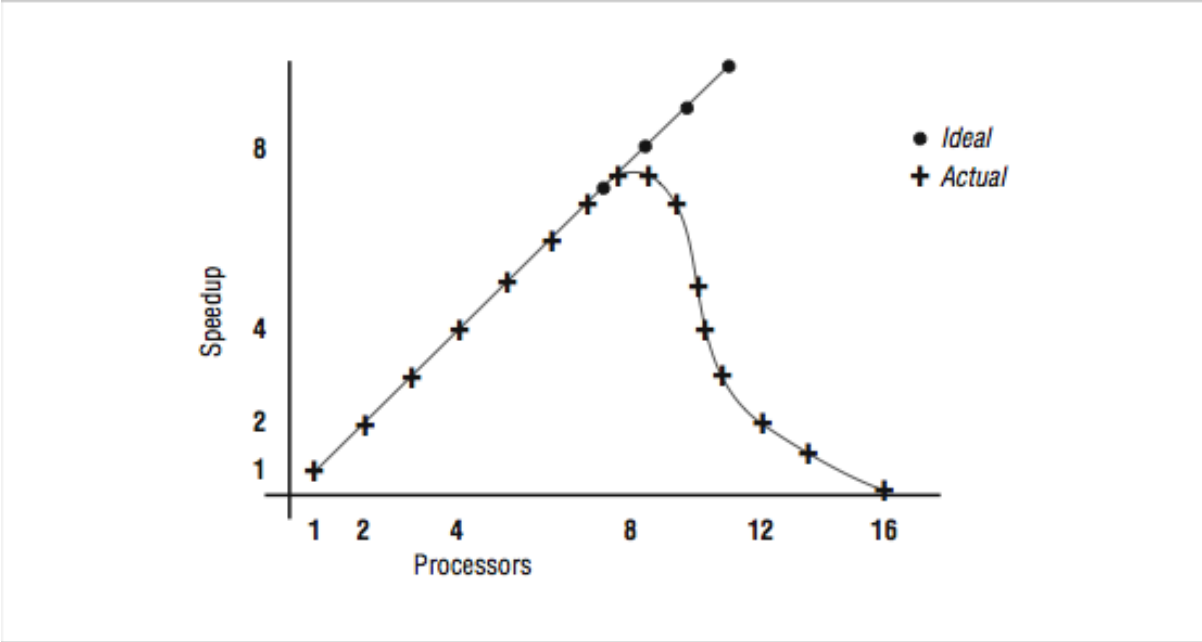
Note that for a while we get nearly perfect speedup, but we begin to see a measurable drop in speedup at four and eight processors. There are several causes for this. In all parallel applications, there is some portion of the code that can't run in parallel. During those nonparallel times, the other processors are waiting for work and aren't contributing to efficiency. This nonparallel code begins to affect the overall performance as more processors are added to the application.

So you say, "this is more like it!" and immediately try to run with 12 and 16 threads. Now, we see the graph in [link] and the data from [link].

Increasing the number of threads

Processors	Time	Speedup
1	30.9	1.0 *(by definition)*
2	15.6	1.98
4	8.2	3.77
8	4.3	7.19
12	14.2	2.18
16	57.0	0.57

Diminishing returns



What has happened here? Things were going so well, and then they slowed down. We are running this program on a 16-processor system, and there are eight other active threads, as indicated below:

```
E6000:uptime
  4:00pm  up 19 day(s), 37 min(s), 5 users, load
average: 8.00, 8.05, 8.14
E6000:
```

Once we pass eight threads, there are no available processors for our threads. So the threads must be time-shared between the processors, significantly slowing the overall operation. By the end, we are executing 16 threads on eight processors, and our performance is slower than with one thread. So it is important that you don’t create too many threads in these types of applications.

Compiler Considerations

Improving performance by turning on automatic parallelization is an example of the “smarter compiler” we discussed in earlier chapters. The addition of a single compiler flag has triggered a great deal of analysis on the part of the compiler including:

- Which loops can execute in parallel, producing the exact same results as the sequential executions of the loops? This is done by checking for dependencies that span iterations. A loop with no interiteration dependencies is called a `DOALL` loop.
- Which loops are worth executing in parallel? Generally very short loops gain no benefit and may execute more slowly when executing in parallel. As with loop unrolling, parallelism always has a cost. It is best used when the benefit far outweighs the cost.
- In a loop nest, which loop is the best candidate to be parallelized? Generally the best performance occurs when we parallelize the outermost loop of a loop nest. This way the overhead associated with beginning a parallel loop is amortized over a longer parallel loop duration.
- Can and should the loop nest be interchanged? The compiler may detect that the loops in a nest can be done in any order. One order may work very well for parallel code while giving poor memory performance. Another order may give unit stride but perform poorly with multiple threads. The compiler must analyze the cost/benefit of each approach and make the best choice.
- How do we break up the iterations among the threads executing a parallel loop? Are the iterations short with uniform duration, or long with wide variation of execution time? We will see that there are a number of different ways to accomplish this. When the programmer has given no guidance, the compiler must make an educated guess.

Even though it seems complicated, the compiler can do a surprisingly good job on a wide variety of codes. It is not magic, however. For example, in the following code we have a loop-carried flow dependency:

```
      PROGRAM DEP
```

```
      PARAMETER(NITER=300,N=1000000)
      REAL*4 A(N)

      DO ITIME=1,NITER
        CALL WHATEVER(A)
        DO I=2,N
            A(I) = A(I-1) + A(I) * C
        ENDDO
      ENDDO
      END
```

When we compile the code, the compiler gives us the following message:

```
      E6000: f77 -O3 -autopar -loopinfo -o dep
dep.f
      dep.f:
      "dep.f", line 6: not parallelized, call may
be unsafe
      "dep.f", line 8: not parallelized, unsafe
dependence (a)
      E6000:
```

The compiler throws its hands up in despair, and lets you know that the loop at Line 8 had an unsafe dependence, and so it won't automatically parallelize the loop. When the code is executed below, adding a thread does not affect the execution performance:

```
      E6000:setenv PARALLEL 1
      E6000:/bin/time dep

      real       18.1
      user       18.1
```

```
sys          0.0
E6000:setenv PARALLEL 2
E6000:/bin/time dep

real        18.3
user        18.2
sys          0.0
E6000:
```

A typical application has many loops. Not all the loops are executed in parallel. It’s a good idea to run a profile of your application, and in the routines that use most of the CPU time, check to find out which loops are not being parallelized. Within a loop nest, the compiler generally chooses only one loop to execute in parallel.

Other Compiler Flags

In addition to the flags shown above, you may have other compiler flags available to you that apply across the entire program:

- You may have a compiler flag to enable the automatic parallelization of reduction operations. Because the order of additions can affect the final value when computing a sum of floating-point numbers, the compiler needs permission to parallelize summation loops.
- Flags that relax the compliance with IEEE floating-point rules may also give the compiler more flexibility when trying to parallelize a loop. However, you must be sure that it’s not causing accuracy problems in other areas of your code.
- Often a compiler has a flag called “unsafe optimization” or “assume no dependencies.” While this flag may indeed enhance the performance of an application with loops that have dependencies, it almost certainly produces incorrect results.

There is some value in experimenting with a compiler to see the particular combination that will yield good performance across a variety of

applications. Then that set of compiler options can be used as a starting point when you encounter a new application.

Assisting the Compiler

If it were all that simple, you wouldn’t need this book. While compilers are extremely clever, there is still a lot of ways to improve the performance of your code without sacrificing its portability. Instead of converting the whole program to C and using a thread library, you can assist the compiler by adding compiler directives to our source code.

Compiler directives are typically inserted in the form of stylized FORTRAN comments. This is done so that a nonparallelizing compiler can ignore them and just look at the FORTRAN code, sans comments. This allows to you tune your code for parallel architectures without letting it run badly on a wide range of single-processor systems.

There are two categories of parallel-processing comments:

- Assertions
- Manual parallelization directives

Assertions tell the compiler certain things that you as the programmer know about the code that it might not guess by looking at the code. Through the assertions, you are attempting to assuage the compiler’s doubts about whether or not the loop is eligible for parallelization. When you use directives, you are taking full responsibility for the correct execution of the program. You are telling the compiler what to parallelize and how to do it. You take full responsibility for the output of the program. If the program produces meaningless results, you have no one to blame but yourself.

Assertions

In a previous example, we compiled a program and received the following output:

```
E6000: f77 -O3 -autopar -loopinfo -o dep dep.f
dep.f:
"dep.f", line 6: not parallelized, call may be
```

```
unsafe
"dep.f", line 8: not parallelized, unsafe
dependence (a)
E6000:
```

An uneducated programmer who has not read this book (or has not looked at the code) might exclaim, “What unsafe dependence, I never put one of those in my code!” and quickly add a *no dependencies* assertion. This is the essence of an assertion. Instead of telling the compiler to simply parallelize the loop, the programmer is telling the compiler that its conclusion that there is a dependence is incorrect. Usually the net result is that the compiler does indeed parallelize the loop.

We will briefly review the types of assertions that are typically supported by these compilers. An assertion is generally added to the code using a stylized comment.

No dependencies

A **no dependencies** or **ignore dependencies** directive tells the compiler that references don’t overlap. That is, it tells the compiler to generate code that may execute incorrectly if there **are** dependencies. You’re saying, “I know what I’m doing; it’s OK to overlap references.” A no dependencies directive might help the following loop:

```
DO I=1,N
  A(I) = A(I+K) * B(I)
ENDDO
```

If you know that `k` is greater than `-1` or less than `-n`, you can get the compiler to parallelize the loop:

```
C$ASSERT NO_DEPENDENCIES
      DO I=1,N
        A(I) = A(I+K) * B(I)
      ENDDO
```

Of course, blindly telling the compiler that there are no dependencies is a prescription for disaster. If `k` equals `-1`, the example above becomes a recursive loop.

Relations

You will often see loops that contain some potential dependencies, making them bad candidates for a no dependencies directive. However, you may be able to supply some local facts about certain variables. This allows partial parallelization without compromising the results. In the code below, there are two potential dependencies because of subscripts involving k and j:

```
for (i=0; i<n; i++) {
    a[i] = a[i+k] * b[i];
    c[i] = c[i+j] * b[i];
}
```

Perhaps we know that there are no conflicts with references to `a[i]` and `a[i+k]`. But maybe we aren't so sure about `c[i]` and `c[i+j]`. Therefore, we can't say in general that there are no dependencies. However, we may be able to say something explicit about `k` (like "`k` is always greater than `-1`"), leaving `j` out of it. This information about the relationship of one expression to another is called a **relation assertion**. Applying a relation assertion allows the compiler to apply its optimization to the first statement in the loop, giving us partial parallelization.[footnote]

Notice that, if you were tuning by hand, you could split this loop into two: one parallelizable and one not.

Again, if you supply inaccurate testimony that leads the compiler to make unsafe optimizations, your answer may be wrong.

Permutations

As we have seen elsewhere, when elements of an array are indirectly addressed, you have to worry about whether or not some of the subscripts may be repeated. In the code below, are the values of K(I) all unique? Or are there duplicates?

```
DO I=1,N
  A(K(I)) = A(K(I)) + B(I) * C
END DO
```

If you know there are no duplicates in `K` (i.e., that `A(K(I))` is a permutation), you can inform the compiler so that iterations can execute in parallel. You supply the information using a *permutation assertion*.

No equivalences

Equivalenced arrays in FORTRAN programs provide another challenge for the compiler. If any elements of two equivalenced arrays appear in the same loop, most compilers assume that references could point to the same memory storage location and optimize very conservatively. This may be true even if it is abundantly apparent to you that there is no overlap whatsoever.

You inform the compiler that references to equivalenced arrays are safe with a *no equivalences* assertion. Of course, if you don’t use equivalences,

this assertion has no effect.

Trip count

Each loop can be characterized by an average number of iterations. Some loops are never executed or go around just a few times. Others may go around hundreds of times:

```
C$ASSERT TRIPCOUNT>100
     DO I=L,N
       A(I) = B(I) + C(I)
     END DO
```

Your compiler is going to look at every loop as a candidate for unrolling or parallelization. It's working in the dark, however, because it can't tell which loops are important and tries to optimize them all. This can lead to the surprising experience of seeing your runtime go up after optimization!

A *trip count assertion* provides a clue to the compiler that helps it decide how much to unroll a loop or when to parallelize a loop.[footnote] Loops that aren't important can be identified with low or zero trip counts. Important loops have high trip counts.
The assertion is made either by hand or from a profiler.

Inline substitution

If your compiler supports procedure inlining, you can use directives and command-line switches to specify how many nested levels of procedures you would like to inline, thresholds for procedure size, etc. The vendor will have chosen reasonable defaults.

Assertions also let you choose subroutines that you think are good candidates for inlining. However, subject to its thresholds, the compiler may reject your choices. Inlining could expand the code so much that increased memory activity would claim back gains made by eliminating the procedure call. At higher optimization levels, the compiler is often capable of making its own choices for inlining candidates, provided it can find the source code for the routine under consideration.

Some compilers support a feature called *interprocedural analysis*. When this is done, the compiler looks across routine boundaries for its data flow analysis. It can perform significant optimizations across routine boundaries, including automatic inlining, constant propagation, and others.

No side effects

Without interprocedural analysis, when looking at a loop, if there is a subroutine call in the middle of the loop, the compiler has to treat the subroutine as if it will have the worst possible side effects. Also, it has to assume that there are dependencies that prevent the routine from executing simultaneously in two different threads.

Many routines (especially functions) don’t have any side effects and can execute quite nicely in separate threads because each thread has its own private call stack and local variables. If the routine is meaty, there will be a great deal of benefit in executing it in parallel.

Your computer may allow you to add a directive that tells you if successive sub-routine calls are independent:

```
C$ASSERT NO_SIDE_EFFECTS
      DO I=1,N
        CALL BIGSTUFF (A,B,C,I,J,K)
      END DO
```

Even if the compiler has all the source code, use of common variables or equivalences may mask call independence.

Manual Parallelism

At some point, you get tired of giving the compiler advice and hoping that it will reach the conclusion to parallelize your loop. At that point you move into the realm of manual parallelism. Luckily the programming model provided in FORTRAN insulates you from much of the details of exactly how multiple threads are managed at runtime. You generally control explicit parallelism by adding specially formatted comment lines to your source code. There are a wide variety of formats of these directives. In this section, we use the syntax that is part of the OpenMP (see [link]) standard. You generally find similar capabilities in each of the vendor compilers. The precise syntax varies slightly from vendor to vendor. (That alone is a good reason to have a standard.)

The basic programming model is that you are executing a section of code with either a single thread or multiple threads. The programmer adds a directive to summon additional threads at various points in the code. The most basic construct is called the *parallel region*.

Parallel regions

In a parallel region, the threads simply appear between two statements of straight-line code. A very trivial example might be the following using the OpenMP directive syntax:

```
        PROGRAM ONE
        EXTERNAL OMP_GET_THREAD_NUM,
OMP_GET_MAX_THREADS
        INTEGER OMP_GET_THREAD_NUM,
OMP_GET_MAX_THREADS
        IGLOB = OMP_GET_MAX_THREADS()
```

```
        PRINT *,'Hello There'
C$OMP PARALLEL PRIVATE(IAM), SHARED(IGLOB)
        IAM = OMP_GET_THREAD_NUM()
        PRINT *, 'I am ', IAM, ' of ', IGLOB
C$OMP END PARALLEL
        PRINT *,'All Done'
        END
```

The `C$OMP` is the sentinel that indicates that this is a directive and not just another comment. The output of the program when run looks as follows:

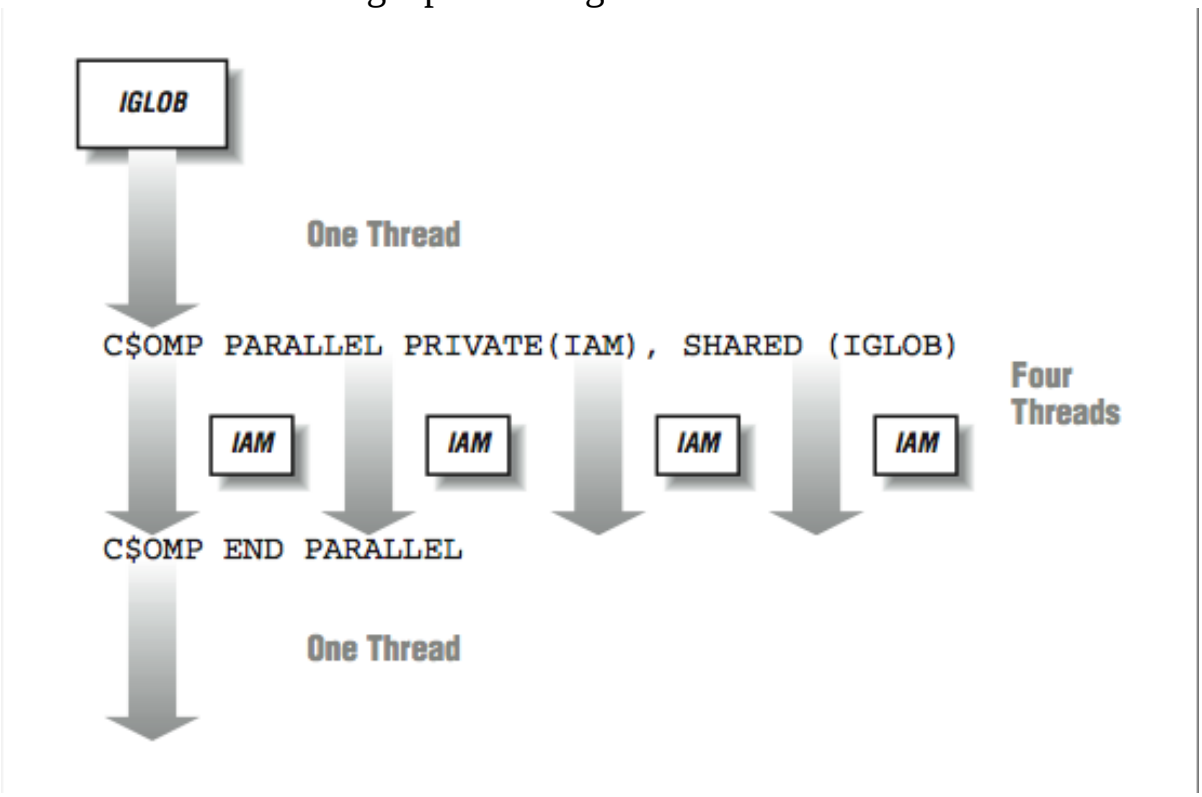
```
% setenv OMP_NUM_THREADS 4
% a.out
Hello There
I am 0 of 4
I am 3 of 4
I am 1 of 4
I am 2 of 4
All Done
%
```

Execution begins with a single thread. As the program encounters the `PARALLEL` directive, the other threads are activated to join the computation. So in a sense, as execution passes the first directive, one thread becomes four. Four threads execute the two statements between the directives. As the threads are executing independently, the order in which the print statements are displayed is somewhat random. The threads wait at the `END PARALLEL` directive until all threads have arrived. Once all threads have completed the parallel region, a single thread continues executing the remainder of the program.

In [link], the `PRIVATE(IAM)` indicates that the `IAM` variable is not shared across all the threads but instead, each thread has its own private version of

the variable. The `IGLOB` variable is shared across all the threads. Any modification of `IGLOB` appears in all the other threads instantly, within the limitations of the cache coherency.

Data interactions during a parallel region



During the parallel region, the programmer typically divides the work among the threads. This pattern of going from single-threaded to multithreaded execution may be repeated many times throughout the execution of an application.

Because input and output are generally not thread-safe, to be completely correct, we should indicate that the print statement in the parallel section is only to be executed on one processor at any one time. We use a directive to indicate that this section of code is a critical section. A lock or other synchronization mechanism ensures that no more than one processor is executing the statements in the critical section at any one time:

```
C$OMP CRITICAL
```

```
        PRINT *, 'I am ', IAM, ' of ', IGLOB
C$OMP END CRITICAL
```

Parallel loops

Quite often the areas of the code that are most valuable to execute in parallel are loops. Consider the following loop:

```
DO I=1,1000000
  TMP1 = ( A(I) ** 2 ) + ( B(I) ** 2 )
  TMP2 = SQRT(TMP1)
  B(I) = TMP2
ENDDO
```

To manually parallelize this loop, we insert a directive at the beginning of the loop:

```
C$OMP PARALLEL DO
      DO I=1,1000000
        TMP1 = ( A(I) ** 2 ) + ( B(I) ** 2 )
        TMP2 = SQRT(TMP1)
        B(I) = TMP2
      ENDDO
C$OMP END PARALLEL DO
```

When this statement is encountered at runtime, the single thread again summons the other threads to join the computation. However, before the threads can start working on the loop, there are a few details that must be handled. The `PARALLEL DO` directive accepts the data classification and

scoping clauses as in the parallel section directive earlier. We must indicate which variables are shared across all threads and which variables have a separate copy in each thread. It would be a disaster to have `TMP1` and `TMP2` shared across threads. As one thread takes the square root of `TMP1`, another thread would be resetting the contents of `TMP1`. `A(I)` and `B(I)` come from outside the loop, so they must be shared. We need to augment the directive as follows:

```
C$OMP PARALLEL DO SHARED(A,B) PRIVATE(I,TMP1,TMP2)
      DO I=1,1000000
        TMP1 = ( A(I) ** 2 ) + ( B(I) ** 2 )
        TMP2 = SQRT(TMP1)
        B(I) = TMP2
      ENDDO
C$OMP END PARALLEL DO
```

The iteration variable I also must be a thread-private variable. As the different threads increment their way through their particular subset of the arrays, they don't want to be modifying a global value for I.

There are a number of other options as to how data will be operated on across the threads. This summarizes some of the other data semantics available:

- **Firstprivate** These are thread-private variables that take an initial value from the global variable of the same name immediately before the loop begins executing.
- **Lastprivate** These are thread-private variables except that the thread that executes the last iteration of the loop copies its value back into the global variable of the same name.
- **Reduction** This indicates that a variable participates in a reduction operation that can be safely done in parallel. This is done by forming a partial reduction using a local variable in each thread and then combining the partial results at the end of the loop.

Each vendor may have different terms to indicate these data semantics, but most support all of these common semantics. [link] shows how the different types of data semantics operate.

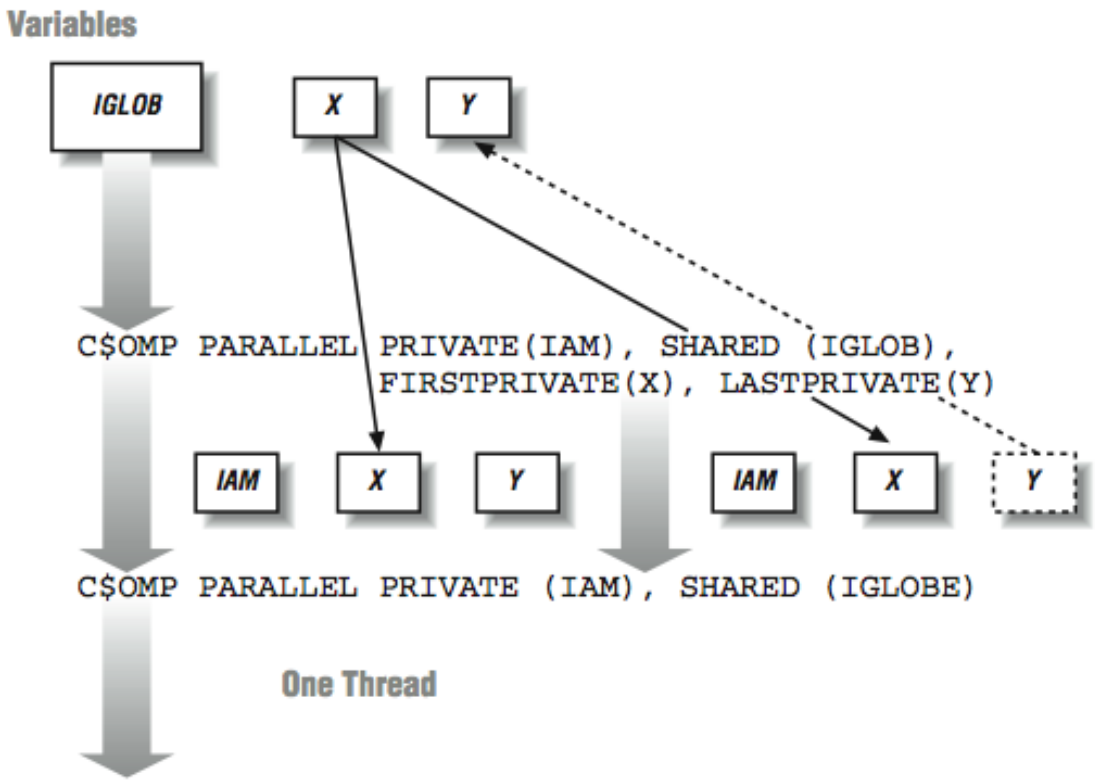
Now that we have the data environment set up for the loop, the only remaining problem that must be solved is which threads will perform which iterations. It turns out that this is not a trivial task, and a wrong choice can have a significant negative impact on our overall performance.

Iteration scheduling

There are two basic techniques (along with a few variations) for dividing the iterations in a loop between threads. We can look at two extreme examples to get an idea of how this works:

```
C VECTOR ADD
      DO IPROB=1,10000
         A(IPROB) = B(IPROB) + C(IPROB)
      ENDDO

C PARTICLE TRACKING
       DO IPROB=1,10000
          RANVAL = RAND(IPROB)
          CALL ITERATE_ENERGY(RANVAL) ENDDO
       ENDDO
```

Variables during a parallel region



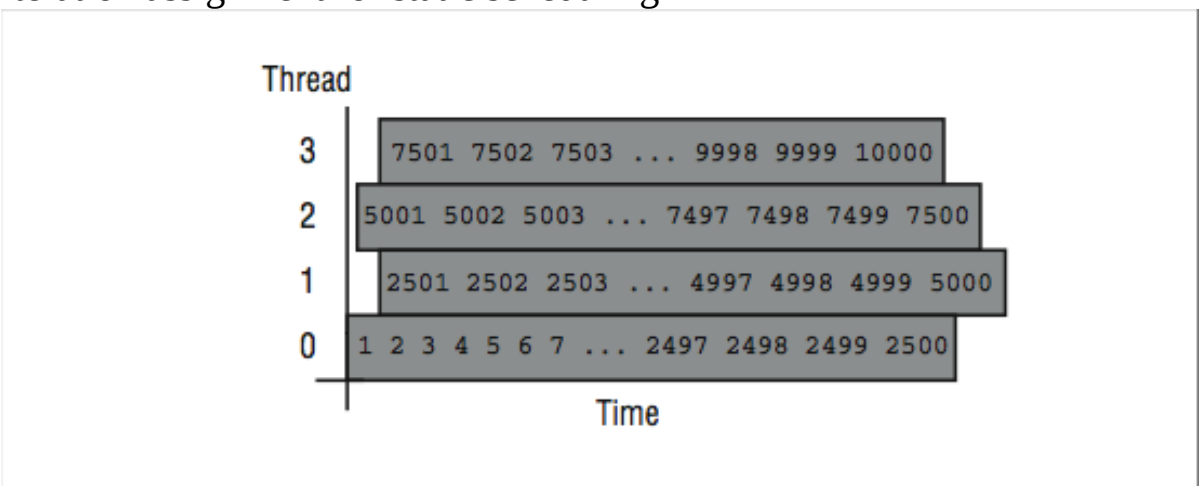
In both loops, all the computations are independent, so if there were 10,000 processors, each processor could execute a single iteration. In the vector-add example, each iteration would be relatively short, and the execution time would be relatively constant from iteration to iteration. In the particle tracking example, each iteration chooses a random number for an initial particle position and iterates to find the minimum energy. Each iteration takes a relatively long time to complete, and there will be a wide variation of completion times from iteration to iteration.

These two examples are effectively the ends of a continuous spectrum of the iteration scheduling challenges facing the FORTRAN parallel runtime environment:

Static
At the beginning of a parallel loop, each thread takes a fixed continuous portion of iterations of the loop based on the number of threads executing the loop.

Dynamic
With dynamic scheduling, each thread processes a chunk of data and when it has completed processing, a new chunk is processed. The chunk size can be varied by the programmer, but is fixed for the duration of the loop.

These two example loops can show how these iteration scheduling approaches might operate when executing with four threads. In the vector-add loop, static scheduling would distribute iterations 1–2500 to Thread 0, 2501–5000 to Thread 1, 5001–7500 to Thread 2, and 7501–10000 to Thread 3. In [link], the mapping of iterations to threads is shown for the static scheduling option.
Iteration assignment for static scheduling



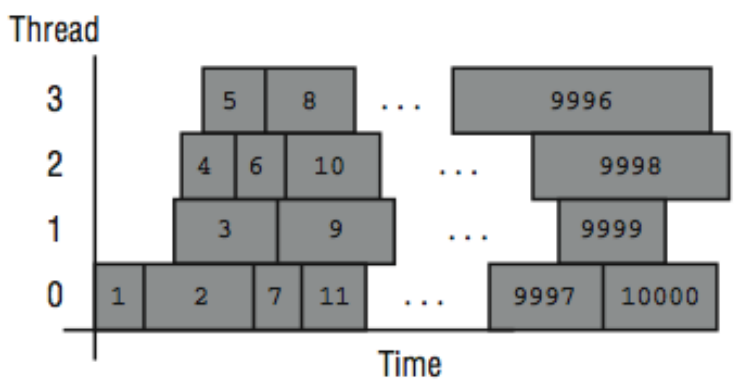
Since the loop body (a single statement) is short with a consistent execution time, static scheduling should result in roughly the same amount of overall work (and time if you assume a dedicated CPU for each thread) assigned to each thread per loop execution.

An advantage of static scheduling may occur if the entire loop is executed repeatedly. If the same iterations are assigned to the same threads that happen to be running on the same processors, the cache might actually contain the values for `A`, `B`, and `C` from the previous loop execution. [footnote] The runtime pseudo-code for static scheduling in the first loop might look as follows:
The operating system and runtime library actually go to some lengths to try to make this happen. This is another reason not to have more threads than available processors, which causes unnecessary context switching.

```
C VECTOR ADD - Static Scheduled
      ISTART = (THREAD_NUMBER * 2500 ) + 1
      IEND = ISTART + 2499
      DO ILOCAL = ISTART,IEND
        A(ILOCAL) = B(ILOCAL) + C(ILOCAL)
      ENDDO
```

It's not always a good strategy to use the static approach of giving a fixed number of iterations to each thread. If this is used in the second loop example, long and varying iteration times would result in poor load balancing. A better approach is to have each processor simply get the next value for `IPROB` each time at the top of the loop.

That approach is called *dynamic scheduling*, and it can adapt to widely varying iteration times. In [link], the mapping of iterations to processors using dynamic scheduling is shown. As soon as a processor finishes one iteration, it processes the next available iteration in order.
Iteration assignment in dynamic scheduling



If a loop is executed repeatedly, the assignment of iterations to threads may vary due to subtle timing issues that affect threads. The pseudo-code for the dynamic scheduled loop at runtime is as follows:

```
C PARTICLE TRACKING - Dynamic Scheduled
     IPROB = 0
     WHILE (IPROB <= 10000 )
       BEGIN_CRITICAL_SECTION
         IPROB = IPROB + 1
         ILOCAL = IPROB
       END_CRITICAL_SECTION
       RANVAL = RAND(ILOCAL)
       CALL ITERATE_ENERGY(RANVAL)
     ENDWHILE
```

`ILOCAL` is used so that each thread knows which iteration is currently processing. The `IPROB` value is altered by the next thread executing the critical section.

While the dynamic iteration scheduling approach works well for this particular loop, there is a significant negative performance impact if the programmer were to use the wrong approach for a loop. For example, if the dynamic approach were used for the vector-add loop, the time to process the critical section to determine which iteration to process may be larger than the time to actually process the iteration. Furthermore, any cache affinity of the data would be effectively lost because of the virtually random assignment of iterations to processors.

In between these two approaches are a wide variety of techniques that operate on a chunk of iterations. In some techniques the chunk size is fixed, and in others it varies during the execution of the loop. In this approach, a chunk of iterations are grabbed each time the critical section is executed. This reduces the scheduling overhead, but can have problems in producing a balanced execution time for each processor. The runtime is modified as follows to perform the particle tracking loop example using a chunk size of 100:

```
      IPROB = 1
      CHUNKSIZE = 100
      WHILE (IPROB <= 10000 )
        BEGIN_CRITICAL_SECTION
          ISTART = IPROB
          IPROB = IPROB + CHUNKSIZE
        END_CRITICAL_SECTION
        DO ILOCAL = ISTART,ISTART+CHUNKSIZE-1
          RANVAL = RAND(ILOCAL)
          CALL ITERATE_ENERGY(RANVAL)
        ENDDO
      ENDWHILE
```

The choice of chunk size is a compromise between overhead and termination imbalance. Typically the programmer must get involved through directives in order to control chunk size.

Part of the challenge of iteration distribution is to balance the cost (or existence) of the critical section against the amount of work done per invocation of the critical section. In the ideal world, the critical section would be free, and all scheduling would be done dynamically. Parallel/vector supercomputers with hardware assistance for load balancing can nearly achieve the ideal using dynamic approaches with relatively small chunk size.

Because the choice of loop iteration approach is so important, the compiler relies on directives from the programmer to specify which approach to use. The following example shows how we can request the proper iteration scheduling for our loops:

```
C VECTOR ADD
C$OMP PARALLEL DO PRIVATE(IPROB) SHARED(A,B,C)
SCHEDULE(STATIC)
      DO IPROB=1,10000
         A(IPROB) = B(IPROB) + C(IPROB)
```

```
      ENDDO
C$OMP END PARALLEL DO
C PARTICLE TRACKING
C$OMP PARALLEL DO PRIVATE(IPROB,RANVAL)
SCHEDULE(DYNAMIC)
      DO IPROB=1,10000
         RANVAL = RAND(IPROB)
         CALL ITERATE_ENERGY(RANVAL)
      ENDDO
C$OMP END PARALLEL DO
```

Closing Notes

Using data flow analysis and other techniques, modern compilers can peer through the clutter that we programmers innocently put into our code and see the patterns of the actual computations. In the field of high performance computing, having great parallel hardware and a lousy automatic parallelizing compiler generally results in no sales. Too many of the benchmark rules allow only a few compiler options to be set.

Physicists and chemists are interested in physics and chemistry, not computer science. If it takes 1 hour to execute a chemistry code without modifications and after six weeks of modifications the same code executes in 20 minutes, which is better? Well from a chemist's point of view, one took an hour, and the other took 1008 hours and 20 minutes, so the answer is obvious.[footnote] Although if the program were going to be executed thousands of times, the tuning might be a win for the programmer. The answer is even more obvious if it again takes six weeks to tune the program every time you make a modification to the program.
On the other hand, if the person is a computer scientist, improving the performance might result in anything from a poster session at a conference to a journal article! This makes for lots of intra-departmental masters degree projects.

In some ways, assertions have become less popular than directives. This is due to two factors: (1) compilers are getting better at detecting parallelism even if they have to rewrite some code to do so, and (2) there are two kinds of programmers: those who know exactly how to parallelize their codes and those who turn on the "safe" auto-parallelize flags on their codes. Assertions fall in the middle ground, somewhere between where the programmer does not want to control all the details but kind of feels that the loop can be parallelized.

You can get online documentation of the OpenMP syntax used in these examples at www.openmp.org.

Exercises

Exercise:

Problem:

Take a static, highly parallel program with a relative large inner loop. Compile the application for parallel execution. Execute the application increasing the threads. Examine the behavior when the number of threads exceed the available processors. See if different iteration scheduling approaches make a difference.

Exercise:

Problem:

Take the following loop and execute with several different iteration scheduling choices. For chunk-based scheduling, use a large chunk size, perhaps 100,000. See if any approach performs better than static scheduling:

```
DO I=1,4000000
  A(I) = B(I) * 2.34
ENDDO
```

Exercise:

Problem:

Execute the following loop for a range of values for N from 1 to 16 million:

```
DO I=1,N
  A(I) = B(I) * 2.34
ENDDO
```

Run the loop in a single processor. Then force the loop to run in parallel. At what point do you get better performance on multiple processors? Do the number of threads affect your observations?

Exercise:

Problem:

Use an explicit parallelization directive to execute the following loop in parallel with a chunk size of 1:

```
      J = 0
C$OMP PARALLEL DO PRIVATE(I) SHARED(J)
SCHEDULE(DYNAMIC)
      DO I=1,1000000
       J = J + 1
      ENDDO
      PRINT *, J
C$OMP END PARALLEL DO
```

Execute the loop with a varying number of threads, including one. Also compile and execute the code in serial. Compare the output and execution times. What do the results tell you about cache coherency? About the cost of moving data from one cache to another, and about critical section costs?

Soporte del Lenguaje para Mejorar el Rendimiento - Introducción

Este capítulo discute los lenguajes de programación que se usan en los sistemas de procesamiento paralelo más grandes. Usualmente, cuando se tope con la necesidad de portar y afinar su código para una nueva arquitectura escalable, tendrá que sentarse durante algún tiempo a replantear su aplicación. A veces requerirá de hacer cambios fundamentales a su algoritmo antes de que pueda comenzar a usar la nueva arquitectura. No se sorprenda si requiere reescribir todo o parte de la aplicación en uno de estos lenguajes. Las modificaciones sobre un sistema bien pueden no reportar beneficios de rendimiento sobre otro sistema distinto. Pero si la aplicación es lo suficientemente importante, no importará el esfuerzo con tal de mejorar el rendimiento.

En este capítulo cubriremos:

- FORTRAN 90
- HPF: High Performance FORTRAN

Estos lenguajes fueron diseñados para usarse en sistemas de cómputo de alto rendimiento. Crearemos paso a paso un pequeño programa en cada uno de ellos, usando un sencillo cálculo de diferencias finitas que modela aproximadamente un flujo calórico. Se trata de un problema clásico que contiene una gran cantidad de paralelismo, y que puede resolverse fácilmente en una gran variedad de arquitecturas paralelas.

Introduciremos y discutiremos el concepto de SPMD (Simple Program, Multiple Data, Programa Único Múltiples Datos), mediante el cual tratamos a las computadoras MIMD como si fueran SIMD. Escribiremos nuestras aplicaciones tal como si estuviéramos usando un gran sistema SIMD para resolver el problema. Pero en vez de usar realmente tal sistema SIMD, la aplicación resultante se compilará para usarse en uno MIMD. La sincronización implícita de los sistemas SIMD será reemplazada en los sistemas MIMD por sincronización explícita a tiempo de ejecución.

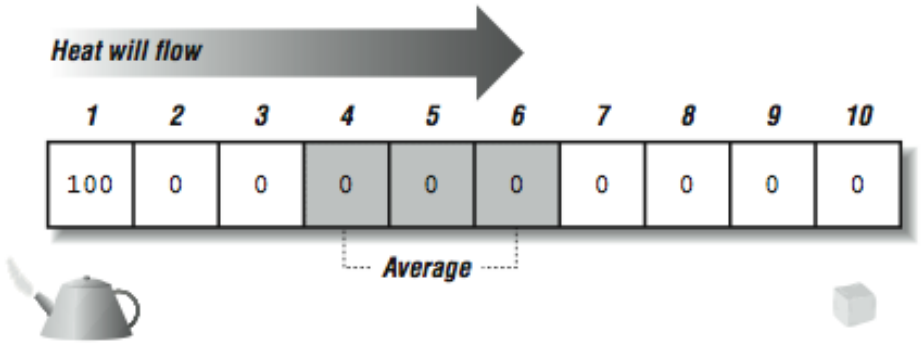
Soporte del Lenguaje para Mejorar el Rendimiento - Problema de Datos en Paralelo: Flujo Calórico

Un problema clásico que explota el procesamiento paralelo escalable, es la simulación de un flujo calórico. La física tras este fenómeno descansa en una ecuación diferencial en derivadas parciales.

Comenzaremos con una placa de metal unidimensional (también conocida como una barra), y avanzaremos hacia una placa bidimensional en los ejemplos posteriores. Al arranque, la barra se encuentra a cero grados Celsius. Luego metemos uno de sus extremos en vapor a 100 grados, y el otro extremo en hielo a cero grados. Queremos simular cómo es el flujo calórico de un extremo al otro, y observar las temperaturas resultantes a lo largo de los distintos puntos de la barra de metal, hasta que dichas temperaturas se estabilicen.

Para lograrlo, dividiremos la barra en 10 segmentos, y seguiremos la pista a la temperatura de cada segmento a lo largo del tiempo. Podemos ver intuitivamente que, en cada ciclo de la simulación, la temperatura que adquirirá cada porción de la barra será el promedio de las temperaturas a su alrededor. Dado que hay temperaturas fijas en ciertos puntos de la barra, el resto eventualmente convergerán a un estado estable tras un número suficiente de ciclos. La [link] muestra el estado al inicio de la simulación.

Flujo Calórico en una Barra



Una implementación simplista de tal simulación es como sigue:

```
      PROGRAM HEATROD
      PARAMETER(MAXTIME=200)
      INTEGER TICKS,I,MAXTIME
      REAL*4 ROD(10)
      ROD(1) = 100.0
      DO I=2,9
        ROD(I) = 0.0
      ENDDO
      ROD(10) = 0.0
      DO TICKS=1,MAXTIME
        IF ( MOD(TICKS,20) .EQ. 1 ) PRINT
100,TICKS,(ROD(I),I=1,10)
        DO I=2,9
          ROD(I) = (ROD(I-1) + ROD(I+1) ) / 2
        ENDDO
      ENDDO
100   FORMAT(I4,10F7.2)
      END
```

La salida de este programa será la siguiente:

```
% f77 heatrod.f
heatrod.f:
 MAIN heatrod:
% a.out
   1 100.00   0.00   0.00   0.00   0.00   0.00
0.00   0.00   0.00  0.00
  21 100.00  87.04  74.52  62.54  51.15  40.30
29.91  19.83   9.92  0.00
  41 100.00  88.74  77.51  66.32  55.19  44.10
33.05  22.02  11.01  0.00
  61 100.00  88.88  77.76  66.64  55.53  44.42
```

```
33.31  22.21  11.10  0.00
  81 100.00  88.89  77.78  66.66  55.55  44.44
33.33  22.22  11.11  0.00
 101 100.00  88.89  77.78  66.67  55.56  44.44
33.33  22.22  11.11  0.00
 121 100.00  88.89  77.78  66.67  55.56  44.44
33.33  22.22  11.11  0.00
 141 100.00  88.89  77.78  66.67  55.56  44.44
33.33  22.22  11.11  0.00
 161 100.00  88.89  77.78  66.67  55.56  44.44
33.33  22.22  11.11  0.00
 181 100.00  88.89  77.78  66.67  55.56  44.44
33.33  22.22  11.11  0.00
%
```

Claramente, para el ciclo de ejecución 101, la simulación converge con una precisión de dos decimales, conforme los números han dejado de cambiar. Debe tratarse de una aproximación a un estado de temperaturas estables al centro de cada segmento de la barra.

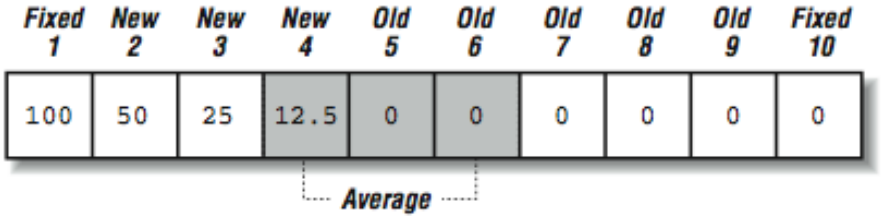
Ahora bien, para este momento los lectores más astutos estarán diciéndose: "Ey, por si no te has dado cuenta, este ciclo tiene una dependencia de flujo." Puede usted alegar que ello impedirá paralelizarlo siquiera un poquito. ¡Es tan grave que ni siquiera puede usted deshacer el bucle para lograr algo de paralelismo a nivel de instrucciones!

Una persona familiarizada con la teoría del flujo calórico también pudiera señalar que el ciclo anterior no implementa *exactamente* el modelo del flujo calórico. El problema es que los valores en el lado derecho de la asignación en el ciclo ROD, se supone que vienen del ciclo de ejecución anterior, y que el valor en el lado izquierdo es el que tendrá en el siguiente ciclo. Pero debido a la manera en que está escrito, el valor ROD(I-1) viene desde el siguiente ciclo, como se muestra en la [link].

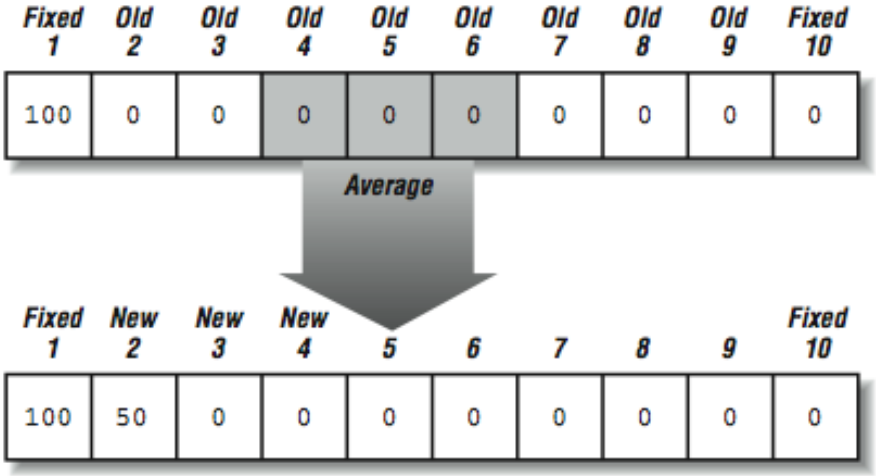
Este problema puede solucionarse usando una técnica conocida como *rojo-negro*, en la cual alternamos entre dos arreglos. La [link] muestra cómo opera (usando dos arreglos para eliminar una dependencia) la versión rojo-

negro del cálculo. ¡Mata dos pájaros de una sola pedrada! Ahora las matemáticas son precisamente correctas, *y* no hay recurrencia. Suena como una situación realmente de ganar-ganar.

Calculando el nuevo valor de una celda.



Usando dos arreglos para eliminar una dependencia



La única desventaja de este enfoque, es que toma el doble de almacenamiento en memoria, y el doble de ancho de banda de memoria. [footnote] El código código modificado es como sigue:

Hay otro enfoque rojo-negro que calcula primero los elementos pares y después los elementos nones de la barra, en dos pasadas. Este enfoque no presenta dependencias de datos entre ambas pasadas. El arreglo ROD nunca tiene todos los valores en el mismo ciclo de ejecución. O bien los valores pares o bien los nones están un ciclo de tiempo adelante de los otros. Se requiere de saltar los elementos de dos en dos y se duplica el ancho de banda necesario, pero no la memoria requerida para resolver el problema.

```
      PROGRAM HEATRED
      PARAMETER(MAXTIME=200)
      INTEGER TICKS,I,MAXTIME
      REAL*4 RED(10),BLACK(10)

      RED(1) = 100.0
      BLACK(1) = 100.0
      DO I=2,9
        RED(I) = 0.0
      ENDDO
      RED(10) = 0.0
      BLACK(10) = 0.0

      DO TICKS=1,MAXTIME,2
        IF ( MOD(TICKS,20) .EQ. 1 ) PRINT
100,TICKS,(RED(I),I=1,10)
        DO I=2,9
          BLACK(I) = (RED(I-1) + RED(I+1) ) / 2
        ENDDO
        DO I=2,9
          RED(I) = (BLACK(I-1) + BLACK(I+1) ) / 2
        ENDDO
      ENDDO
100   FORMAT(I4,10F7.2)
      END
```

La salida del programa modificado es como sigue:

```
% f77 heatred.f
heatred.f:
 MAIN heatred:
% a.out
```

```
   1 100.00   0.00   0.00   0.00   0.00   0.00
0.00   0.00   0.00  0.00
  21 100.00  82.38  66.34  50.30  38.18  26.06
18.20  10.35   5.18  0.00
  41 100.00  87.04  74.52  61.99  50.56  39.13
28.94  18.75   9.38  0.00
  61 100.00  88.36  76.84  65.32  54.12  42.91
32.07  21.22  10.61  0.00
  81 100.00  88.74  77.51  66.28  55.14  44.00
32.97  21.93  10.97  0.00
 101 100.00  88.84  77.70  66.55  55.44  44.32
33.23  22.14  11.07  0.00
 121 100.00  88.88  77.76  66.63  55.52  44.41
33.30  22.20  11.10  0.00
 141 100.00  88.89  77.77  66.66  55.55  44.43
33.32  22.22  11.11  0.00
 161 100.00  88.89  77.78  66.66  55.55  44.44
33.33  22.22  11.11  0.00
 181 100.00  88.89  77.78  66.67  55.55  44.44
33.33  22.22  11.11  0.00
%
```

Resulta interesante notar que el programa modificado requiere de más tiempo de convergencia que la primera versión. Converge hasta la iteración 181, en vez de en la 101. Si observa usted la primera versión, notará que debido a la recurrencia, el calor terminaba fluyendo más rápido de izquierda a derecha, dado que el elemento izquierdo de cada promedio era el valor del siguiente ciclo de ejecución. Puede parecer ágil, pero está mal. [footnote] Generalmente, en este problema, cualquier enfoque converge a los mismos valores eventuales dentro de los límites de la representación de punto flotante.

Existen otros enfoques algorítmicos para resolver las ecuaciones diferenciales en derivadas parciales, tales como el "método rápido multipolo" que acelera la convergencia "legalmente". No asuma que el enfoque de fuerza bruta usado aquí es el único método para resolver este problema en particular. Los programadores siempre deben buscar el mejor

algoritmo disponible (sea o no paralelo) antes de intentar escalar el algoritmo "equivocado". Para aquellos cuates que no son científicos en computación, el tiempo para la solución es más importante que el incremento lineal de velocidad.

El problema de flujo calórico es extremadamente simple, y en su forma rojo-negro, resulta inherentemente muy paralelizable con interacciones muy simples de datos. Resulta un buen modelo para una amplia variedad de problemas en los cuales discretizamos espacios bidimensionales y tridimensionales, y realizamos alguna simulación sencilla en tales espacios.

Este problema usualmente puede escalarse creando una malla más fina. A menudo, el beneficio de usar procesadores escalables viene de permitir crear dichas mallas más finas, y no de obtener soluciones en tiempos más cortos. Por ejemplo, puede que usted sea capaz de realizar una simulación climática a nivel mundial, usando una malla con una separación de 200 millas, con un solo procesador en cuatro horas. Usando 100 procesadores, puede que sea capaz de ejecutar la misma simulación usando un tamaño de malla de 20 millas, también en cuatro horas, pero con resultados mucho más precisos. O, utilizando 400 procesadores, puede hacer una simulación de malla aún más fina en una hora.

Soporte del Lenguaje para Mejorar el Rendimiento - Lenguajes Explícitamente Paralelos

Tal como hemos visto a lo largo de este libro, uno de los mayores retos de afinación consiste en que el compilador reconozca aquellos segmentos de código particulares que pueden paralelizarse. Esto resulta especialmente cierto para códigos numéricos, donde el retorno de inversión es potencialmente mayor. Piense en esto: si usted sabe que algo es paralelo, ¿por qué debiera haber alguna dificultad en hacer que el compilador lo reconozca? ¿Por qué no puede usted simplemente escribirlo, y hacer que el compilador diga "Sí, esto se hará en paralelo"?

El problema es que los lenguajes de programación más comúnmente usados no ofrecen ningún constructo para expresar cálculos en paralelo. Usted se ve forzado a expresarse en términos primitivos, como si fuera un cavernícola con grandes ideas, pero sin vocabulario para explicarlas. Esto es particularmente cierto en FORTRAN y en C, que no soportan la noción de cómputos en paralelo, lo cuál significa que los programadores deben reducir los cálculos a pasos secuenciales. Suena incómodo, pero la mayoría de los programadores lo hacen de forma tan natural que ni siquiera se percatan de cuán buenos son para ello.

Por ejemplo, digamos que queremos sumar dos vectores, A y B. ¿Cómo debemos hacerlo? Probablemente escribamos un pequeño ciclo sin siquiera detenernos un instante a pensarlo:

```
DO I=1,N
  C(I) = A(I) + B(I)
END DO
```

Parece razonable, pero veamos qué pasa. ¡Le hemos impuesto un orden a los cálculos! ¿No hubiera bastado con decir "C es el resultado de A más B"? Ello permitiría al compilador sumar los vectores usando cualesquiera hardware a su disposición, usando el método que mejor le parezca. De eso

se tratan los lenguajes paralelos, de proporcionar primitivas capaces de expresar cómputos paralelos.

No se están proponiendo nuevos lenguajes paralelos tan rápidamente como se hizo a mediados de la década de 1980. Los desarrolladores se han dado cuenta que pueden crear un esquema maravilloso, pero si no es compatible con FORTRAN o C, le va a interesar a poca gente. La razón es simple: hay miles de millones de líneas de código C y FORTRAN, pero sólo unas pocas en *Fizgibbet* o como sea que le llame usted a su nuevo lenguaje paralelo. Por causa de la preminencia de C y FORTRAN, las actividades más significativas referentes a lenguajes paralelos que se desarrollan actualmente se enfocan a extender estos lenguajes, protegiendo así 20 o 30 años de inversión en programas ya escritos. [footnote] Resulta muy tentador para los desarrolladores de un nuevo lenguaje probarlo mediante el problema de las ocho reinas y el juego de la vida, obtener buenos resultados, y luego declarar que está listo para su debut y esperar que hordas de programadores lo conviertan en su lenguaje de cabecera.
Uno de los esfuerzos más significativos en el campo de los lenguajes completamente nuevos es SISAL (Streams and Iteration in a Single Assignment Language). Es un lenguaje de flujo de datos que se puede integrar fácilmente en módulos de FORTRAN y C. Los aspectos más interesantes de SISAL son la cantidad de código que se ha portado a SISAL, y el hecho de que los desarrolladores de SISAL generalmente comparan su rendimiento con el de aplicaciones iguales en FORTRAN y C.

Soporte del Lenguaje para Mejorar el Rendimiento - FORTRAN 90

La versión previa del estandar de FORTRAN liberada por ANSI (el American National Standard Institute), FORTRAN 77 (X3.9-1978) se escribió para promover la portabilidad de los programas FORTRAN entre plataformas distintas. No se inventaron nuevos componentes del lenguaje, más bien se incorporaron buenas características que ya estaban disponibles en compiladores en ambientes de producción. Al contrario que FORTRAN 77, el FORTRAN 90 (ANSI X3.198-1992) añade nuevas extensiones y características al lenguaje. Algunas de ellas sólo ponen al día a FORTRAN respecto a lenguajes como C (asignación dinámica de memoria, reglas de alcance) y C++ (interfaces de funciones con genericidad). Pero algunas de estas nuevas características son únicas de FORTRAN (operaciones sobre arreglos). Es interesante notar que mientras se desarrollaba la especificación de FORTRAN 90, las arquitecturas de cómputo de alto rendimiento dominantes eran los sistemas SIMD escalables como la Connection Machine, así como los sistemas de procesadores vectoriales paralelos de compañías como Cray Research.

FORTRAN 90 hace un trabajo sorprendentemente bueno en cubrir las necesidades de estas arquitecturas tan diferentes. Sus características también se adaptan razonablemente bien a los nuevos multiprocesadores con memoria compartida uniforme. Sin embargo, como veremos después, FORTRAN 90 por sí solo todavía no es suficiente para cumplir las necesidades de los sistemas distribuidos escalables con acceso a memoria no uniforme, que se están volviendo dominantes en la computación de extremo superior.

Las extensiones de FORTRAN 90 a FORTRAN 77 incluyen:

- Constructos para arreglos
- Asignación dinámica de memoria y variables automáticas
- Apuntadores
- Nuevos tipos de datos, estructuras
- Nuevas funciones intrínsecas, incluyendo muchas que operan sobre vectores y matrices
- Nuevas estructuras de control, tales como la sentencia WHERE
- Interfaces de procedimientos mejoradas

Constructos para Arreglos en FORTRAN 90

Con los constructos para arreglos de FORTRAN 90, puede usted especificar arreglos completos o secciones de los mismos como participantes en operaciones unarias y binarias. Tales constructos son una característica clave para "deserializar" las aplicaciones, de forma que puedan adaptarse mejor a computadoras vectoriales o procesadores paralelos. Por ejemplo, digamos que desea usted sumar dos vectores A y B. En FORTRAN 90, puede expresarlo como una simple operación de suma, en vez de usando el bucle tradicional. Esto es, puede usted escribir:

```
A = A + B
```

en vez del tradicional bucle en FORTRAN 77:

```
DO I=1,N
  A(I) = A(I) + B(I)
ENDDO
```

Puede que el código generado por el compilador en su estación de trabajo no se vea muy diferente, pero en algunas máquinas de arquitectura paralela o en algunas estaciones de trabajo a la vuelta de la esquina, la diferencia es significativa. La versión FORTRAN 90 declara explícitamente que los cálculos pueden realizarse en cualquier orden, incluyendo todos en paralelo simultáneamente.

Un efecto importante de ello es que si la versión de FORTRAN 90 experimenta una falla de punto flotante sumando el elemento 17, y revisa usted la memoria en un depurador, encontrará que el elemento 27 ya fue calculado y tiene un valor perfectamente válido.

Y no está usted limitado tan sólo a arreglos unidimensionales. Por ejemplo, la suma a nivel de elementos de dos arreglos bidimensionales puede realizarse así:[footnote]
Sólo en caso de que esté usted sorprendido, A*B regresa el producto de los miembros del arreglo a nivel de elementos, no el producto de dos matrices. Tal operación la cubre FORTRAN 90 mediante una función intrínseca.

```
A = A + B
```

en vez de:

```
DO J=1,M
  DO I=1,N
    A(I,J) = A(I,J) + B(I,J)
  END DO
END DO
```

Naturalmente, si desea combinar dos arreglos en una operación, sus tamaños tienen que ser compatibles. Sumar un vector de siete elementos a uno de ocho no tiene sentido. Tampoco lo tiene multiplicar un arreglo de 2*4 a uno de 3*4.. Cuando los dos arreglos tienen tamaños compatibles, relativos a la operación a que quiere usted someterlos, decimos que tienen *concordancia de tamaños*, como en el siguiente código:

```
DOUBLE PRECISION A(8), B(8)
 ...
A = A + B
```

Siempre se considera que los escalares tienen concordancia de tamaños con los arreglos (y con otros escalares). En una operación binaria con un arreglo, un escalar se trata como un arreglo del mismo tamaño, con el único elemento duplicado en el resto de las posiciones.

Aun así, existen limitaciones. Cuando haga referencia a un arreglo particular, por ejemplo A, está haciendo referencia a todo él, del primer al último elemento. Seguro puede usted imaginar casos donde le interese especificar un subconjunto de un arreglo. Puede tratarse de un grupo de elementos consecutivos o algo como "cada octavo elemento" (i.e. un recorrido de salto no unitario a través del arreglo). A tales partes de los arreglos, posiblemente no contiguas, se les denomina *secciones del arreglo*.

Para especificar secciones de arreglos en FORTRAN 90, se reemplazan los índices tradicionales con tripletas de la forma `a:b:c`, que significan "los elementos desde `a` hasta `b`, tomados con un incremento de `c`." Pueden omitirse partes de la tripleta, siempre y cuando el significado permanezca claro. Por ejemplo, `a:b` significa "los elementos desde a hasta `b`;" `a:` significa "los elementos desde `a` hasta el extremo superior del arreglo con un incremento de 1." Recuerdo que una tripleta reemplaza a un único índice, así que un arreglo *n*-dimensional puede tener *n* tripletas.

Puede usted usar tripletas en expresiones, nuevamente asegurándose de que las partes de las expresiones estén en concordancia. Considere estos ejemplos:

```
REAL X(10,10), Y(100)
    ...
X(10,1:10)   = Y(91:100)
X(10,:)      = Y(91:100)
```

La primera sentencia asigna los últimos 10 elementos de `Y` al décimo renglón de `X`. La segunda sentencia expresa lo mismo de modo ligeramente diferente. El " : " solitario le indica al compilador que está implícito el rango completo (del 1 al 10).

Intrínsecos en FORTRAN 90

FORTRAN 90 extiende la funcionalidad de los intrínsecos de FORTRAN 77, y además agrega muchos nuevos, incluyendo algunas subrutinas intrínsecas. La mayoría pueden ser *valoradas mediante arreglos*: pueden regresar secciones de arreglos o escalares, dependiendo de cómo fueron invocadas. Por ejemplo, he aquí un nuevo uso del intrínseco `SIN` valorado mediante arreglos:

```
REAL A(100,10,2)
 ...
A = SIN(A)
```

Cada elemento del arreglo A es reemplazado por su seno. Los intrínsecos de FORTRAN 90 también trabajan con secciones de arreglos, con tal de que la variable que reciba los resultados esté en concordancia de tamaño con la que se pasó:

```
REAL A(100,10,2)
REAL B(10,10,100)
 ...
B(:,:,1) = COS(A(1:100:10,:,1))
```

También se han extendido otros intrínsecos tales como `SQRT`, `LOG`, etc. Entre los nuevos intrínsecos se encuentran:

- **Reducciones** FORTRAN 90 tiene reducciones de vectores tales como `MAXVAL`, `MINVAL` y `SUM`. Para arreglos de orden superior (cualquier cosa mayor que un vector) tales funciones pueden realizar una reducción a lo largo de una dimensión en particular. Adicionalmente, existe una función `DOT_PRODUCT` (producto punto) para los vectores.

- **Manipulación de matrices** Los intrínsecos `MATMUL` y `TRANSPOSE` pueden manipular matrices completas.
- **Construir o redimensionar arreglos**`RESHAPE` le permite crear un nuevo arreglo a partir de los elementos de uno antiguo con diferente tamaño. `SPREAD` replica un arreglo a lo largo de una nueva dimensión. `MERGE` copia porciones de un arreglo en otro, bajo el control de una máscara. `CSHIFT` permite desplazar un arreglo en una o más dimensiones.
- **Funciones de consulta** `SHAPE`, `SIZE`, `LBOUND`, y `UBOUND` le permiten averiguar cómo está construido un arreglo.
- **Pruebas en paralelo** Dos intrínsecos nuevos de reducción, `ANY` y `ALL`, sirven para probar en paralelo varios elementos de un arreglo.

Nuevas Características de Control

FORTRAN 90 incluye algunas nuevas características de control, entre ellas una *primitiva de asignación* condicional llamada `WHERE`, que pone las asignaciones de un arreglo con concordancia de tamaños bajo el control de una máscara, tal como en el ejemplo siguiente. He aquí un ejemplo de la primitiva `WHERE`:

```
REAL A(2,2), B(2,2), C(2,2)
DATA B/1,2,3,4/, C/1,1,5,5/
 ...
WHERE (B .EQ. C)
  A = 1.0
  C = B + 1.0
ELSEWHERE
  A = -1.0
ENDWHERE
```

En aquellos casos donde la expresión lógica es `TRUE`, `A` obtiene `1.0` y `C` obtiene `B+1.0`. En la claúsula `ELSEWHERE`, `A` obtiene `-1.0`. El resultado

de la operación anterior serán los arreglos `A` y `C` con los elementos:

```
A =  1.0  -1.0          C =  2.0    5.0
    -1.0  -1.0               1.0    5.0
```

Nuevamente, no hay ningún orden implícito en estas asignaciones condicionales, lo cual significa que pueden realizarse en paralelo. La ausencia de orden implícito es crítica para permitir a los sistemas de cómputo SIMD y a los ambientes SPMD tener flexibilidad en la ejecución de tales cálculos.

Arreglos Automáticos y Asignables

Cada programa requiere variables temporales o espacio de trabajo. En el pasado, los programadores de FORTRAN a menudo administraban su propio espacio libre, declarando un arreglo lo suficientemente grande para manejar cualquier requerimiento temporal. Esta práctica se traga la memoria (aunque la memoria virtual, por lo general), e incluso puede tener efectos indeseados en el rendimiento. Con la habilidad de asignar la memoria dinámicamente, los programadores pueden esperar más tiempo para decidir cuánto espacio vacío deben obtener. FORTRAN 90 soporta la asignación dinámica de memoria mediante dos nuevas características del lenguaje: arreglos automáticos y arreglos asignables.

Tal como sucede con las variables locales de un programa en C, los arreglos automáticos de FORTRAN 90 sólo tienen asignado espacio de almacenamiento durante la vida de la subrutina o función que los contiene. Esto es diferente del almacenamiento local de arreglos tradicional de FORTRAN, donde algo de espacio se apartaba a tiempo de compilación o enlace. El tamaño y forma de los arreglos automáticos puede esculpirse mediante ua combinación de constantes y argumentos. Por ejemplo, he aquí una declaración de un arreglo automático, `B`, usando la nueva especificación de sintaxis de FORTRAN 90:

```
      SUBROUTINE RELAX(N,A)
      INTEGER N
      REAL, DIMENSION (N) :: A, B
```

Se declararon dos arreglos: `A`, el argumento de prueba, y `B`, un arreglo automático de tamaño explícito. Cuando se regresa de la subrutina, `B` deja de existir. Observe que el tamaño de `B` se toma de uno de los argumentos, `N`.

Los arreglos asignables le dan a usted la flexibilidad de elegir el tamaño de un arreglo después de examinar otras variables en el programa. Por ejemplo, puede que desee determinar la cantidad de datos de entrada antes de asignar los arreglos. Este pequeño programa pregunta al usuario el tamaño de la matriz antes de asignarle espacio de almacenamiento:

```
      INTEGER M,N
      REAL, ALLOCATABLE, DIMENSION (:,:) :: X
       ...
      WRITE (*,*) 'INTRODUZCA LAS DIMENSIONES DE
X'
      READ (*,*) M,N
      ALLOCATE (X(M,N))
       ...
       hacer algo con X
       ...
      DEALLOCATE (X)
       ...
```

La sentencia `ALLOCATE` crea un arreglo de `M × N` que posteriormente se libera mediante la sentencia `DEALLOCATE`. Tal como sucede con los programas en `C`, es importante devolver la memoria asignada cuando se ha

terminado de usar; de otra forma, su programa pudiera consumir toda la memoria virtual de almacenamiento disponible.

Flujo Calórico en FORTRAN 90

El problema de flujo calórico es un programa ideal para demostrar cuán agradablemente puede expresar FORTRAN 90 los problemas que emplean arreglos regulares:

```
              PROGRAM HEATROD
              PARAMETER(MAXTIME=200)
              INTEGER TICKS,I,MAXTIME
              REAL*4 ROD(10)
              ROD(1) = 100.0
              DO I=2,9
                ROD(I) = 0.0
              ENDDO
              ROD(10) = 0.0
              DO TICKS=1,MAXTIME
                IF ( MOD(TICKS,20) .EQ. 1 ) PRINT
100,TICKS,(ROD(I),I=1,10)
                ROD(2:9) = (ROD(1:8) + ROD(3:10) ) /
2
              ENDDO
      100     FORMAT(I4,10F7.2)
              END
```

El programa es idéntico, excepto por el hecho de que el bucle interno ha sido reemplazado por una sola sentencia que calcula la "nueva" sección, al promediar una tira de los elementos "izquierdos" y una tira de los elementos "derechos".

La salida de este programa luce así:

```
E6000: f90 heat90.f
E6000:a.out
   1 100.00   0.00   0.00   0.00   0.00   0.00
0.00   0.00   0.00   0.00
  21 100.00  82.38  66.34  50.30  38.18  26.06
18.20  10.35   5.18   0.00
  41 100.00  87.04  74.52  61.99  50.56  39.13
28.94  18.75   9.38   0.00
  61 100.00  88.36  76.84  65.32  54.12  42.91
32.07  21.22  10.61   0.00
  81 100.00  88.74  77.51  66.28  55.14  44.00
32.97  21.93  10.97   0.00
 101 100.00  88.84  77.70  66.55  55.44  44.32
33.23  22.14  11.07   0.00
 121 100.00  88.88  77.76  66.63  55.52  44.41
33.30  22.20  11.10   0.00
 141 100.00  88.89  77.77  66.66  55.55  44.43
33.32  22.22  11.11   0.00
 161 100.00  88.89  77.78  66.66  55.55  44.44
33.33  22.22  11.11   0.00
 181 100.00  88.89  77.78  66.67  55.55  44.44
33.33  22.22  11.11   0.00
E6000:
```

Si lo observa detenidamente, verá que la salida es la misma que la de la implementación rojo-negro. Ello se debe a que en FORTRAN 90:

```
ROD(2:9) = (ROD(1:8) + ROD(3:10) ) / 2
```

es una *única* sentencia de asignación. Como se muestra en [link], la parte derecha se evalúa completamente antes de que la sección del arreglo

resultante se le asigne a `ROD(2:9)`. A primera vista puede parecer poco natural, pero considere la siguiente sentencia:

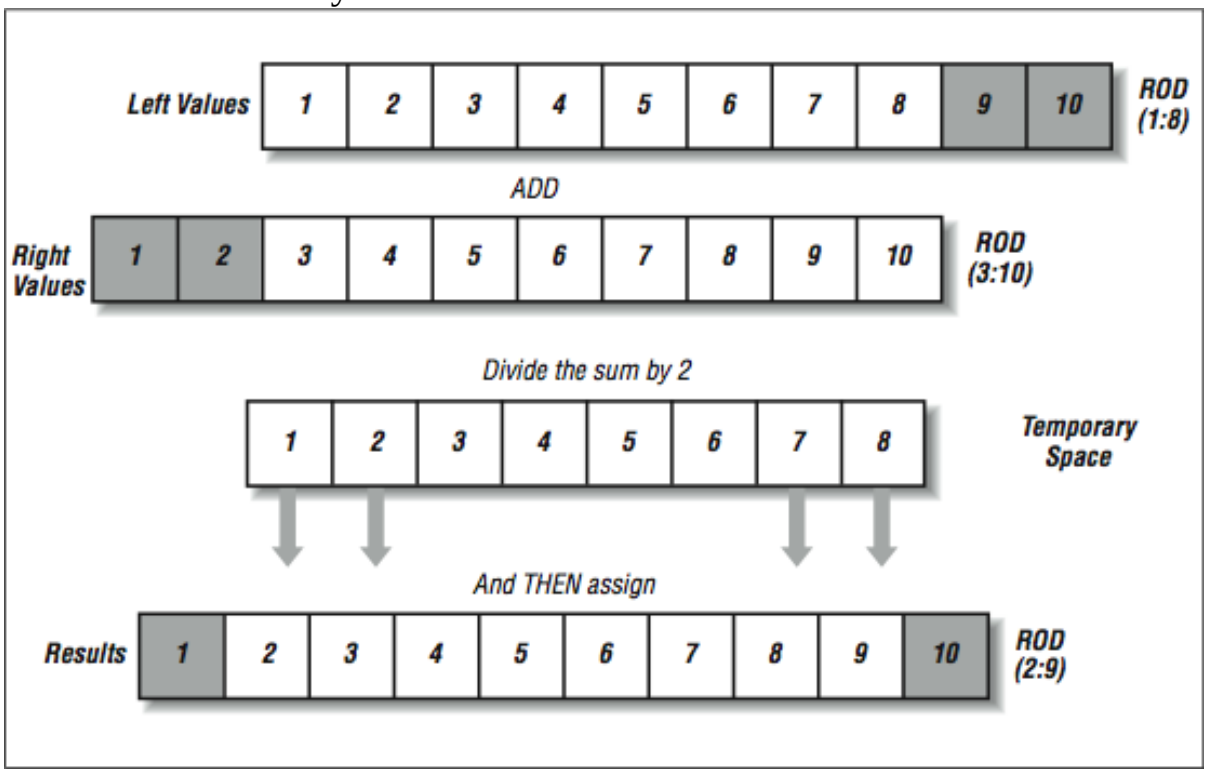
```
I = I + 1
```

Sabemos que si `I` comienza en 5, esta sentencia incrementará su valor a 6. Ello sucede porque el lado derecho (5+1) se evalúa antes de ejecutar la asignación del 6 a `I`. En FORTRAN 90, una variable puede ser un arreglo completo. Así que ésta *is* una operación rojo-negro. ¡Hay un `ROD` "antiguo" en la parte derecha, y un `ROD` "nuevo" en la parte izquierda!

Para realmente "pensar" al estilo FORTRAN 90, es bueno suponer que está usted usando un sistema SIMD con millones de pequeñas CPUs. Primero alineamos cuidadosamente los datos, deslizándolos alrededor, y entonces... ¡zas! en una sola instrucción, sumamos todos los valores alineados en un solo instante. [link] muestra gráficamente este acto de "alinear" los valores y luego sumarlos. El grafo de flujo de datos es extremadamente simple. Los dos renglones superiores son de sólo lectura, y los datos fluyen de arriba hacia abajo. Usar el espacio temporal elimina la dependencia aparente. Este enfoque de "pensar SIMD" es una de dos formas posibles de obligarnos a enfocar nuestro pensamiento en los datos, en vez de en el control. Puede que la SIMD no sea una buena arquitectura para nuestro problema, pero si es capaz de expresarlo de forma tal que pueda trabajarse en SIMD, un buen ambiente SPMD puede tomar ventaja del paralelismo a nivel de datos que usted ha identificado.

El ejemplo siguiente pone de manifiesto uno de los retos que aparecen al producir una implementación eficiente de FORTRAN 90. Si estos arreglos contienen 10 millones de elementos, y el compilador usa un enfoque simple, necesitará 30 millones de elementos para los valores "izquierdos" viejos, los valores "derechos" viejos y para los nuevos. Se requiere optimizar el flujo de datos sólo para determinar para determinar cuántos datos extras deben mantenerse para obtener los resultados apropiados. Si el compilador es inteligente, la memoria extra puede ser muy poca:

Alineación de datos y cálculos




```
SAVE1 = ROD(1)
DO I=2,9
  SAVE2 = ROD(I)
  ROD(I) = (SAVE1 + ROD(I+1) ) / 2
  SAVE1 = SAVE2
ENDDO
```

Si bien esta implementación no tiene el paralelismo de una implementación rojo-negro completa, produce los resultados correctos con sólo dos elementos de datos extras. El truco consiste en guardar el valor "izquierdo" viejo justo antes de borrarlo. Un buen compilador de FORTRAN 90 usa análisis de flujo de datos, en busca de una plantilla sobre cómo se mueven los cálculos a lo largo de los datos para ver si puede guardar unos pocos elementos durante un corto periodo de tiempo, para aliviar la necesidad de una copia extra completa de los datos.

La ventaja del lenguaje FORTRAN 90 es que depende del compilador si usa una copia completa del arreglo, o unos pocos elementos de datos para asegurar que el programa se ejecute apropiadamente. Y lo que es más importante, puede cambiar su enfoque conforme cambia de una arquitectura a otra.

FORTRAN 90 Versus FORTRAN 77

Es interesante señalar que FORTRAN 90 nunca fue completamente adoptado por la comunidad de cómputo de alto rendimiento. Existen algunas razones para ello:

- Existía la preocupación de que el uso de apuntadores y estructuras de datos dinámicas arruinase el rendimiento, y se perdieran las ventajas de optimización de FORTRAN sobre C. Algunos dijeron que FORTRAN 90 estaba tratando de ser un mejor C que C. Otros decían, "¿quién quiere parecerse más a un lenguaje más lento?" Sea cual fuere la razón, hubo algo de controversia cuando se implementó FORTRAN 90, que derivó en algo de reluctancia de los programadores para adoptarlo. Algunos vendedores decían "Puede usted usar FORTRAN 90, pero FORTRAN 77 siempre será más rápido."
- Y como los vendedores a menudo implementaron diferentes subconjuntos de FORTRAN 90, el código no fue tan transportable como el de FORTRAN 77. Por tal motivo, los usuarios que requerían máxima transportabilidad continuaron con FORTRAN 77.
- A veces los vendedores compraban sus compiladores que cumplían completamente con FORTRAN 90 a terceros, quienes cobraban altas tarifas de licenciamiento. Así, podía usted obtener un FORTRAN 77 gratuito (y más rápido, de acuerdo con el vendedor) o pagar por el compilador de FORTRAN 90 más lento (guiño, guiño).
- Por tales factores, el número de aplicaciones serias desarrolladas en FORTRAN 90 fue pequeño. Así que los bancos de pruebas usados para comprar nuevos sistemas estaban desarrollados casi exclusivamente en FORTRAN 77. Ello motivó a los vendedores a mejorar sus compiladores de FORTRAN 77 en vez de sus compiladores de FORTRAN 90.

- Conforme los compiladores de FORTRAN 77 se hacían más sofisticados, usando análisis de flujo de datos, fue relativamente fácil escribir código "paralelo" transportable en FORTRAN 77, usando las técnicas que hemos discutido en este libro.
- Uno de los mayores beneficios potenciales de FORTRAN 90 era la transportabilidad entre SIMD y las supercomputadoras paralelas/vectoriales. Conforme ambas arquitecturas fueron reemplazadas con multiprocesadores con memoria uniforme compartida, FORTRAN 77 se convirtió en el lenguaje que ofrecía la máxima transportabilidad entre las computadoras usadas típicamente por los programadores de cómputo de alto rendimiento.
- Los compiladores de FORTRAN 77 soportaban directivas que permitían a los programadores afinar finamente el rendimiento de sus aplicaciones, al tomar completo control del paralelismo. Ciertos dialectos de FORTRAN 77 esencialmente se convirtieron en un "lenguaje ensamblador" para programación paralela. Incluso las versiones altamente afinadas de tales códigos eran relativamente transportables entre multiprocesadores de memoria compartida uniforme de diferentes vendedores.

Así que los eventos conspiraron contra FORTRAN 90 en el corto plazo. Sin embargo, FORTRAN 77 no está bien adaptado a los sistemas de memoria distribuida porque no se presta bien para las directivas de disposición de datos. Conforme requerimos particionar y distribuir los datos cuidadosamente sobre esos nuevos sistemas, debemos dar al compilador *gran cantidad* de flexibilidad. FORTRAN 90 es el lenguaje mejor adaptado para tal propósito.

Resumen de FORTRAN 90

Bueno, este es una visita apresurada a FORTRAN 90. Probablemente no le hayamos hecho justicia al lenguaje cubriéndolo tan brevemente, pero queremos que usted se lleve una probada del mismo. Hay muchas características que no hemos discutido. Si quiere aprender más, le recomendamos *FORTRAN 90 Explained, de Michael Metcalf y John Reid (Oxford University Press)*.

FORTRAN 90 no es suficiente por sí mismo para que obtengamos rendimiento escalable en sistema de memoria distribuida. Es más, ningún compilador es todavía capaz de realizar suficiente análisis de datos para decidir dónde almacenar los datos y cuándo recuperarlos de la memoria. Así, por ahora, nosotros como programadores debemos preocuparnos por la disposición de los datos. Debemos descomponer el problema en fragmentos paralelos que puedan procesarse individualmente. Tenemos varias opciones. Podemos usar High Performance FORTRAN y delegar algunos detalles al compilador, o podemos usar paso de mensajes explícito y ocuparnos nosotros mismos de *todos* los detalles del paralelismo.

Soporte del Lenguaje para Mejorar el Rendimiento - Descomposición del Problema

Existen tres enfoques principales sobre cómo dividir o descomponer un trabajo, de modo que pueda distribuirse entre múltiples CPUs:

- **Descomposición de los cálculos** Ya hemos discutido esta técnica. Cuando la descomposición se hace basándose en los cálculos, llegamos a algún mecanismo para dividir equitativamente los cómputos (tales como las iteraciones o un bucle) entre nuestros procesadores. Generalmente se ignora la ubicación de los datos, y las preocupaciones principales son la duración de cada iteración y la uniformidad. Esta es la técnica predilecta para los sistemas de memoria uniformemente compartida, porque cualquier procesador puede acceder a los datos de modo igualitario.
- **Decomposición de los datos** Cuando el acceso a la memoria no es uniforme, la tendencia es enfocarse en la distribución de los datos en vez de los cómputos. Se asume que la recuperación de los datos "remotos" es costosa y por tanto debe minimizarse. Los datos se distribuyen entre las memorias. El procesador que contiene el dato realiza los cálculos sobre dicho dato tras recuperar cualquier otro dato necesario para realizarlos.
- **Decomposición de las tareas** Cuando las operaciones que deben realizarse son muy independientes entre sí, y toman algún tiempo, puede llevarse a cabo una descomposición de tareas. En este enfoque un proceso o hilo de ejecución maestro mantiene una cola de las unidades de trabajo. Cuando un procesador dispone de recursos, recupera la siguiente "tarea" de la cola y comienza a procesarla. Se trata de un enfoque muy atractivo para cálculos embarazosamente paralelos.[footnote]
  El esfuerzo distribuido para romper la clave RC5 fue coordinado de esta forma. Cada procesador recibió un bloque de claves y comenzó a probar dichas claves. En algún punto, si los procesadores no eran lo suficientemente rápidos o dejaban de funcionar, el sistema central reasignaba el bloque a otro procesador. Esto permite al sistema recuperarse de problemas en las computadoras individuales.

En cierto sentido, el resto del capítulo está dedicado primordialmente a la descomposición de datos. En un sistema de memoria distribuida, el costo de comunicación usualmente es el factor de rendimiento dominante. Si su problema es tan violentamente paralelo que puede distribuirse entre tareas, prácticamente cualquier técnica funcionará. Los problemas con datos paralelos ocurren en muchas disciplinas. Pueden variar desde aquellos que son extremadamente paralelos, a aquellos otros que sólo lo son un poco. Por ejemplo, los cálculos fractales son extremadamente paralelos; cada punto se deriva independientemente del resto. Es sencillo dividir los cálculos fractales entre los procesadores. Y dado que los cálculos son independientes, los procesadores no requieren coordinarse o compartir datos.

Nuestro problema de flujo calórico, cuando se expresa en su forma rojo-negro (o FORTRAN 90) es extremadamente paralelo, pero requiere de cierto grado de compartición de datos. Un modelo gravitatorio o una galaxia es otra clase de programa paralelo. Cada punto ejerce una influencia sobre cualesquiera otro. Así, y al contrario que los cálculos fractales, los procesadores deben compartir datos.

Sea cual sea el caso, usted quiere acomodar los cálculos de forma que los procesadores puedan decirse los unos a los otros, "empieza a partir de aquí y trabaja en esto, mientras que yo trabajo en esto otro, y lo reuniremos nuevamente cuando hayamos terminado."

Aun los problemas que ofrecen menos independencia entre regiones son todavía muy buenos candidatos para la descomposición del dominio. Los problemas de diferencias finitas, la simulación de la interacción entre partículas próximas, y las columnas de matrices pueden tratarse de forma similar. Si puede usted dividir el dominio igualmente entre los procesadores, cada uno realizará aproximadamente la misma cantidad de trabajo en su camino hacia la solución.

Otros sistemas físicos no son tan regulares, o involucran interacciones de rango aomplio. Los nodos de una rejilla no estructurada pueden no estar acomodados en correspondencia directa con sus ubicaciones físicas, por ejemplo. O tal vez el modelo involucre fuerzas de rango amplio, tales como la atracción entre partículas. Estos problemas también pueden estructurarse

para máquinas paralelas, aunque resulte algo más difícil. A veces se requiere de varias simplificaciones, o de "agrupamiento" de efectos intermedios. Por ejemplo, la influencia de un grupo de partículas distantes sobre otro puede tratarse como si se tratar de partículas compuestas actuando a distancia. Con ello se ahorran las comunicaciones que se requieren que cada procesador haga para hablar con cada uno de los otros, acerca de cada detalle. En otros casos, la arquitectura paralela ofrece oportunidades de expresar un sistema físico en formas distintas e ingeniosas, que tienen sentido en el contexto de la máquina. Por ejemplo, puede asignarse cada partícula a su propio procesador, de forma que puedan deslizarse unas sobre otras, agregando interacciones y actualizando un ciclo de tiempo.

Dependiendo de la arquitectura de la computadora paralela y del problema, elegir entre dividir o replicar (porciones de) el dominio puede agregar una sobrecarga de trabajo o costos inaceptables para todo el proyecto.

En problemas grandes, el costo económico de la memoria principal puede dejar fuera de toda discusión el mantener copias locales separadas del mismo dato. De hecho, a menudo es la necesidad de más memoria lo que conduce a las personas hacia las máquinas paralelas; el problema que requieren resolver no cabe en la memoria de una computadora convencional.

Invirtiendo algo de esfuerzo, puede usted permitir que el particionamiento del dominio evolucione conforme el programa se ejecuta, en respuesta a una distribución de carga dispareja. De esta forma, si hubiera muchas solicitudes para A, entonces varios procesadores podrán obtener dinámicamente una copia de la pieza A del dominio. O bien la pieza A puede estar dispersa entre varios procesadores, cada uno manejando un subconjunto distinto de definiciones de A. Incluso puede usted migrar copias únicas de datos de un lugar a otro, cambiando su residencia conforme se requiera.

Cuando el dominio de datos es irregular, o cambia a lo largo del tiempo, el programa paralelo encuentra un problema de balance de cargas. Tal problema se vuelve esencialmente aparente cuando toma mucho más tiempo en completarse una porción de los cálculos paralelos que otros. Un

ejemplo de la vida real puede ser un análisis de ingeniería de una retícula adaptativa. Conforme se ejecuta el problema, la retícula se vuelve más refinada en aquellas áreas que muestran mayor actividad. Si el trabajo no se reacomoda de vez en cuando, la sección de la computadora responsable de la sección de la retícula con mayor refinamiento sufrirá una caída progresiva del rendimiento respecto al resto de la máquina.

Soporte del Lenguaje para Mejorar el Rendimiento - FORTRAN de Alto Rendimiento (HPF)

En marzo de 1992, el Foro de Fortran de Alto Rendimiento (HPFF por sus siglas en inglés) comenzó a reunirse para discutir y definir un conjunto de agregados a FORTRAN 90 para hacer más práctico su uso en ambientes de cómputo escalable. El plan era desarrollar una especificación durante ese año, de forma que los vendedores rápidamente comenzaran a implementar el estándar. El alcance de este esfuerzo incluía lo siguiente:

- Identificar escalares y arreglos que puedan distribuirse a lo largo de la máquina paralela.
- Indicar cómo se distribuirán. ¿Serán tiras, bloques o de alguna otra forma?
- Especificar cómo se alinearán dichas variables las unas con respecto a las otras.
- Redistribuir y realinear estructuras de datos a tiempo de ejecución.
- Añadir una estructura de control `FORALL` para las asignaciones paralelas que sean difíciles o imposibles de construirse usando la sintaxis de arreglos de FORTRAN.
- Hacerle mejoras a la estructura de control `WHERE` de FORTRAN 90.
- Añadir funciones intrínsecas para las operaciones en paralelo más comunes.

Existían varias fuentes de inspiración para el esfuerzo HPF. Las directivas de disposición ya eran parte del ambiente de programación de FORTRAN 90 en algunas computadoras SIMD (i.e., la CM-2). También se había liberado el año anterior PVM, el primer ambiente de paso de mensajes transportable, y los usuarios tenían un año de experiencia tratando de descomponer programas a mano. Habían desarrollado algunas técnicas básicas utilizables para la descomposición de datos que funcionaban muy bien, pero requerían de mucha más contabilidad.[footnote]
Como pronto veremos.

El esfuerzo HPF agrupó un conjunto diverso de intereses de todos los principales vendedores de cómputo de alto rendimiento. Estaban representadas las principales compañías. Como resultado, HPF se diseñó para implementarse en casi todo tipo de arquitecturas.

Hay un esfuerzo en marcha para producir el siguiente estándar de FORTRAN, FORTRAN 95, que se espera adopte algunas pero no todas las modificaciones de HPF.

Programando en HPF

HPF usa FORTRAN 90 como su núcleo. Si un programa en FORTRAN 90 se pasa por un compilador HPF, debe producir los mismos resultados que si se hubiera compilado con FORTRAN 90. Asumiendo que un programa HPF sólo use los constructos de FORTRAN 90 y las directivas HPF, un compilador FORTRAN 90 puede ignorar las directivas y debe producir los mismos resultados que un compilador HPF.

Conforme el usuario agrega directivas al programa, la semántica de éste no cambia. Si el usuario no entiende en absoluto la aplicación e inserta directivas extremadamente mal pensadas, el programa produce resultados correctos aunque muy lentamente. Un compilador HPF no trata de "mejorar" las directivas del usuario. Asume que el programador es omniscente.[footnote]
Lo cual siempre es bueno de asumirse.

Una vez que el usuario ha determinado cómo distribuir los datos entre los procesadores, el compilador de HPF trata de usar el mínimo de comunicaciones necesario y traslapa la comunicación con los cálculos siempre que sea posible. HPF generalmente usa la regla de "el propietario calcula" para la ubicación de los cálculos. Un elemento particular en un arreglo se calcula mediante el procesador que almacena dicho elemento del arreglo.

Si es necesario, se recogen de procesadores remotos todos los datos necesarios para realizar el cálculo. Si el programador hace una descomposición y alineación inteligentes, muchos de los datos requeridos estarán en la memoria local, en vez de en una memoria remota. El compilador HPF también es responsable de asignar cualquier estructura de datos temporal necesaria para soportar las comunicaciones a tiempo de ejecución.

En general, el compilador HPF no es mágico -simplemente hace un muy buen trabajo con los detalles de comunicación cuando el programador es capaz de diseñar una buena descomposición de datos. Al mismo tiempo, retiene la transportabilidad con máquinas de una sola CPU y sistemas de memoria uniforme compartida usando FORTRAN 90.

Directivas HPF de Disposición de Datos

Tal vez las contribuciones más importantes de HPF sean las directivas de disposición de datos. Usándolas, el programador puede controlar cómo se acomodan los datos, basándose en su conocimiento acerca de las interacciones entre datos. Una directiva de ejemplo es la que sigue:

```
      REAL*4 ROD(10)
!HPF$ DISTRIBUTE ROD(BLOCK)
```

El prefijo `!HPF$` es visto como un comentario por cualquier compilador que no sea HPF, de forma que un compilador FORTRAN 90 común pueda ignorarla de forma segura. La directiva `DISTRIBUTE` indica que el arreglo `ROD` debe distribuirse entre múltiples procesadores. Si no se usase dicha directiva, el arreglo `ROD` se ubicaría en un procesador y se le comunicaría a los otros procesos conforme se necesitara. Existen varios esquemas de distribución que pueden hacerse en cada dimensión:

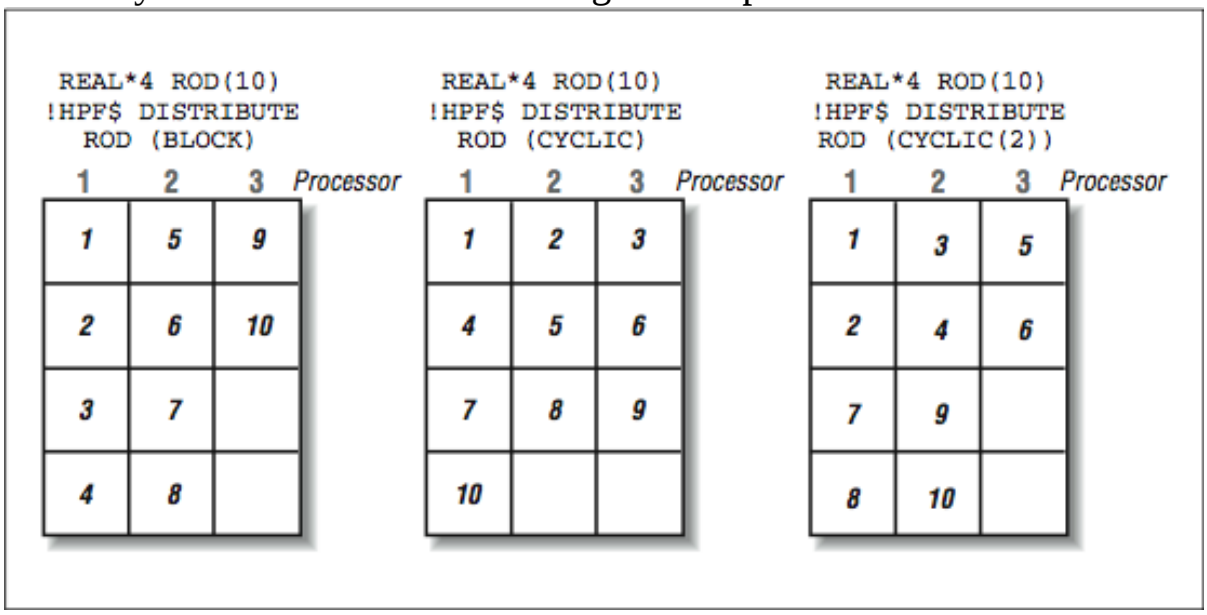
```
      REAL*4
BOB(100,100,100),RICH(100,100,100)
!HPF$ DISTRIBUTE BOB(BLOCK,CYCLIC,*)
!HPF$ DISTRIBUTE RICH(CYCLIC(10))
```

Estas distribuciones operan como sigue:

- `BLOCK` El arreglo se distribuye a lo largo de los procesadores usando bloques contiguos del valor del índice. Los bloques se hacen tan grandes como sea posible.
- `CYCLIC` El arreglo se distribuye a lo largo de los procesadores, mapeando cada elemento sucesivo al "siguiente" procesador, y cuando se llega al último procesador, la ubicación comienza nuevamente en el primero.
- `CYCLIC(n)` El arreglo se distribuye del mismo modo que en `CYCLIC` excepto que se colocan `n` elementos sucesivos en cada procesador antes de pasarse al siguiente.

Note: Todos los elementos en esta dimensión se colocan en el mismo procesador. Esto es mayormente útil para arreglos multidimensionales.

Distribuyendo los elementos del arreglo entre procesadores

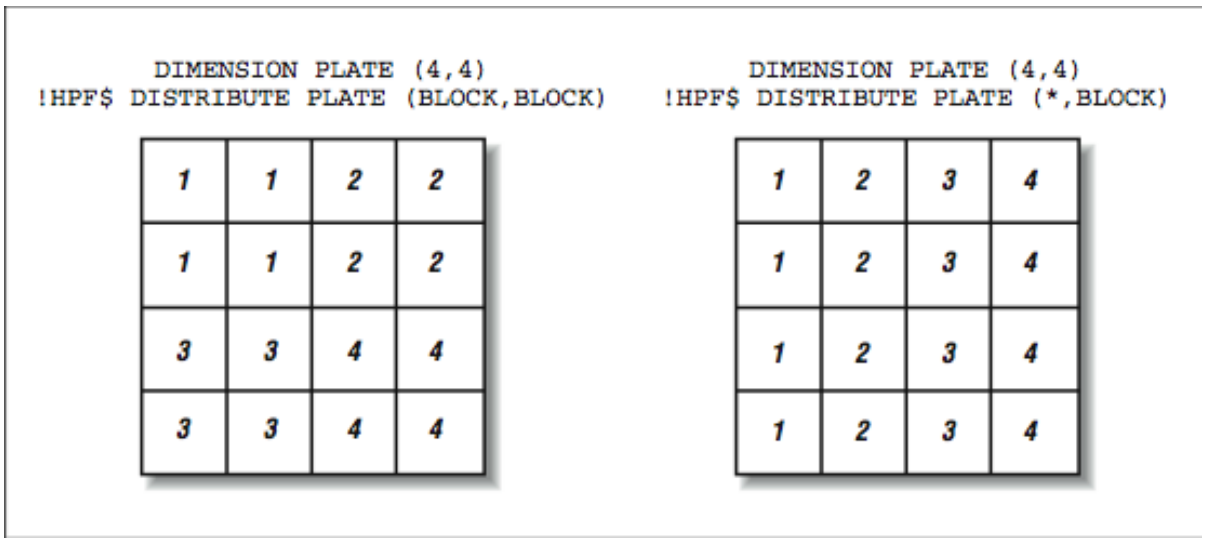


La [link] muestra cómo se mapean los elementos de un arreglo sencillo entre tres procesadores con diferentes directivas.

Deben ubicarse cuatro elementos en los Procesadores 1 y 2 porque no hay un Procesador 4 disponible para el elemento más a la izquierda si se ubican tres elementos en los Procesadores 1 y 2. En [link], los elementos se ubican en procesadores sucesivos, regresando al Procesador 1 tras el último procesador. En [link], usar un tamaño de bocado de `CYCLIC` es un compromiso entre `BLOCK` puro y `CYCLIC` puro.

Para explorar el uso de `*`, debemos observar un simple arreglo bidimensional mapeado entre cuatro procesadores. En [link], mostramos la distribución del arreglo y cada celda indica cuál procesador almacenará el dato para dicha celda en un arreglo bidimensional. En [link], la directiva lo descompone en ambas dimensiones simultáneamente. Este enfoque resulta en unos parches aproximadamente cuadrados en el arreglo. Sin embargo, puede que no sea el mejor enfoque. En el siguiente ejemplo, usamos el `*` para indicar que queremos que todos los elementos de una columna particular sean ubicados sobre le mismo procesador. Así, los valores de columna distribuyen equitativamente las columnas entre los procesadores. Entonces, todos los renglones en cada columna siguen donde ha sido colocada la columna. Ello permite un salto unitario para las porciones ubicadas adentro de los procesadores, y resulta benéfico en algunas aplicaciones. La sintaxis de `*` también se conoce como distribución `sobre el procesador`.

Distribuciones bidimensionales



Cuando lidie con más de una estructura de datos para realizar un cálculo, puede bien sea distribuirlas separadamente , o bien usar la directiva `ALIGN` para asegurarse de que los elementos correspondientes de ambas estructuras estén alojados juntos. En el siguiente ejemplo, tenemos un arreglo de placa y un factor de escala que debemos aplicar a cada columna de la placa durante el cálculo:

```
      DIMENSION PLATE(200,200),SCALE(200)
!HPF$ DISTRIBUTE PLATE(*,BLOCK)
!HPF$ ALIGN SCALE(I) WITH PLATE(J,I)
```

O bien:

```
      DIMENSION PLATE(200,200),SCALE(200)
!HPF$ DISTRIBUTE PLATE(*,BLOCK)
!HPF$ ALIGN SCALE(:) WITH PLATE(*,:)
```

En ambos ejemplos, las variables `PLATE` y `SCALE` están ubicadas en los mismos procesadores que las columnas correspondientes de `PLATE`. La sintaxis `*` y `:` comunican la misma información. Cuando se usa `*`, esa dimensión se colapsa y no participa en la distribución. Cuando se usa `:`, significa que la esa dimensión sigue a la dimensión correspondiente en la variable que ya ha sido distribuida.

También puede usted especificar el acomodo de la variable `SCALE` y hacer que la variable `PLATE` "siga" la distribución de la variable `SCALE`:

```
      DIMENSION PLATE(200,200),SCALE(200)
!HPF$ DISTRIBUTE SCALE(BLOCK)
```

```
!HPF$ ALIGN PLATE(J,I) WITH SCALE(I)
```

Puede agregar expresiones aritméticas simples en la directiva `ALIGN`, sujetas a ciertas limitaciones. Las otras directivas incluyen:

- `PROCESSORS` Le permite crear una forma de configuración de los procesos que pueda usarse para alinear otras estructuras de datos.
- `REDISTRIBUTE y REALIGN` Le permite cambiar dinámicamente la forma de las estructuras de datos a tiempo de ejecución, conforme cambian los patrones de comunicación durante el curso de la misma.
- `TEMPLATE` Le permite crear un arreglo que no usa espacio. En vez de distribuir una estructura de datos y alinear todas las demás, algunos usuarios crearán y distribuirán una plantilla y luego alinearán todas las estructuras de datos reales de acuerdo a esa plantilla.

El uso de directivas puede fluctuar desde lo muy simple a lo muy complejo. En algunas situaciones, usted distribuirá la única estructura grande compartida, alineando unas pocas estructuras relacionadas y habrá terminado. En otras, los programadores intentan optimizar las comunicaciones basándose en la topología de la red de interconexión (hypercubo, red de interconexión multietapa, malla o toroide) usando directivas muy detalladas. También pueden redistribuir cuidadosamente los datos durante las varias fases del cómputo.

Con suerte, su aplicación logrará un buen rendimiento sin demasiado esfuerzo.

Estructuras de control en HPF

Mientras los diseñadores de HPF estaban enmedio de definir un nuevo lenguaje, se dieron a la tarea de mejorar aquello que habían visto como una limitación de FORTRAN 90. Es interesante que tales modificaciones son las que se están considerando como parte del nuevo estándar de FORTRAN 95.

La sentencia `FORALL` permite al usuario expresar operaciones iterativas sencillas que se aplican al arreglo completo, sin descansar en un ciclo do-loop (recuerde, los ciclos do-loop fuerzan un orden). Por ejemplo:

```
FORALL (I=1:100, J=1:100) A(I,J) = I + J
```

Ello puede expresarse en FORTRAN 90 nativo, pero es mucho más feo, contraintuitivo y propenso a errores.

Otra estructura de control proporciona la habilidad de declarar una función como "PURE." Una función PURE no tiene efectos secundarios mas que a través de sus parámetros. El programador garantiza que una función PURE puede ejecutarse simultáneamente en muchos procesadores sin efectos indeseables. Esto le permite a HPF asumir que dicha función sólo operará sobre datos locales y que no requiere de ninguna comunicación de datos durante el tiempo que dure su ejecución completa. El programador también puede declarar cuáles parámetros de la función son de entrada, cuáles de salida y cuáles de ambos.

Intrínsecos de HPF

Las compañías que venden computadoras SIMD requieren entregarlas con herramientas que permitan la ejecución de operaciones colectivas eficientes sobre todos los procesadores. Un ejemplo perfecto de esto es la operación SUM. Para `SUM` el valor de un arreglo a lo largo de N procesadores, el enfoque más simplista toma N pasos. Sin embargo, es posible lograrlo en log(N) pasos usando una técnica denominada *suma prefija en paralelo*. Al momento en que se estaba desarrollando HPF se habían identificado e implementado varias de tales operaciones. HPF dio la oportunidad de definir una sintaxis estandarizada para ellas.

Una muestra de tales operaciones incluye:

- `SUM_PREFIX` Realiza varios tipos de sumas prefijas en paralelo.

- `ALL_SCATTER` Distribuye un único valor a un conjunto de procesadores.
- `GRADE_DOWN` Ordena en orden decreciente.
- `IANY` Calcula el OR lógico de un conjunto de valores.

Aun cuando existe un gran número de tales funciones intrínsecas, la mayoría de las aplicaciones sólo usa unas pocas de tales operaciones.

Extrínsecos de HPF

Con el objeto de permitir a los vendedores de arquitecturas diversas, proporcionar sus ventajas particulares, HPF incluyó la capacidad de enlazarse con funciones "extrínsecas". Tales funciones no requieren de reescribirse en FORTRAN 90/HPF, y realizan varias capacidades sólo soportadas por los vendedores. Esta capacidad permite a los usuarios realizar tales tareas, como la creación de aplicaciones híbridas, con algo de HPF y algo de paso de mensajes.

Los programadores de cómputo de alto rendimiento siempre gustan de tener la habilidad de hacer cosas a su propio modo, con el objetivo de exprimir hasta la última gota de rendimiento.

Flujo Calórico en HPF

Para transportar nuestra aplicación de flujo calórico a HPF, realmente sólo requerimos de agregar una línea de código. En el ejemplo siguiente, lo hemos cambiado a un arreglo bidimensional más grande:

```
        INTEGER PLATESIZ,MAXTIME
        PARAMETER(PLATESIZ=2000,MAXTIME=200)
!HPF$   DISTRIBUTE PLATE(*,BLOCK)
        REAL*4 PLATE(PLATESIZ,PLATESIZ)
        INTEGER TICK
        PLATE = 0.0
```

```
      * Sumar las fronteras
              PLATE(1,:) = 100.0
              PLATE(PLATESIZ,:) = -40.0
              PLATE(:,PLATESIZ) = 35.23
              PLATE(:,1) = 4.5

              DO TICK = 1,MAXTIME
                PLATE(2:PLATESIZ-1,2:PLATESIZ-1) =
(
         +        PLATE(1:PLATESIZ-2,2:PLATESIZ-1)
+
         +        PLATE(3:PLATESIZ-0,2:PLATESIZ-1)
+
         +        PLATE(2:PLATESIZ-1,1:PLATESIZ-2)
+
         +        PLATE(2:PLATESIZ-1,3:PLATESIZ-0)
) / 4.0
                PRINT 1000,TICK, PLATE(2,2)
      1000      FORMAT('TICK = ',I5, F13.8)
              ENDDO
      *
              END
```

Notará que la directiva HPF distribuye las columnas del arreglo usando el enfoque `BLOCK`, manteniendo todos los elementos de una columna sobre un mismo procesador. A primera vista, pudiera parecer que (`BLOCK`,`BLOCK`) es la mejor distribución. Sin embargo, hay dos ventajas de una distribución (`*`,`BLOCK`). Primero, recorrer hacia abajo una columna es una operación de paso único, y de este modo puede usted procesar una columna completa. El aspecto más significativo de la distribución es que una distribución (`BLOCK`,`BLOCK`) fuerza a cada procesador a comunicarse con hasta ocho procesadores más para obtener los valores de sus vecinos. Usando la distribución (*,`BLOCK`), cada procesador tendrá que intercambiar datos con cuando mucho dos procesadores en cada ciclo.

Cuando veamos PVM, veremos el mismo programa implementado al estilo de paso de mensajes SPMD. En este ejemplo, verá usted algunos de los detalles que HFP debe manejar para ejecutar apropiadamente este código. Tras revisarlo ¡probablemente usted elija implementar todas sus aplicaciones futuras de flujo calórico en HPF!

Resumen de HPF

En ciertas cosas HPF ha sido mejor que FORTRAN 90. Compañías como IBM con su SP-1 requieren de proporcionar algún lenguaje de alto nivel para el que sus usuarios no quieren escribir código de paso de mensajes. Por tal motivo, IBM ha invertido una gran cantidad de esfuerzo en implementar un HPF optimizado. Resulta interesante que mucho de su esfuerzo beneficiará directamente la habilidad de desarrollar compiladores de FORTRAN 90 más sofisticados. El extensivo análisis de flujo de datos requerido para minimizar las comunicaciones y administrar las estructuras de datos dinámicos recaerá en los compiladores de FORTRAN 90 incluso sin usar las directivas de HPF.

El tiempo dirá si ya no serán necesarias las directivas de distribución de datos de HPF, y si los compiladores serán capaces de realizar suficiente análisis sobre el código plano de FORTRAN 90 para optimizar la ubicación y movimiento de los datos.

En su forma actual, HPF es un vehículo excelente para expresar las aplicaciones altamente paralelas a nivel de datos, y basadas en retículas. Sus debilidades son las comunicaciones irregulares y el balance de cargas dinámicas. Un nuevo esfuerzo para desarrollar la siguiente versión de HPF está en camino, para manejar algunos de estos temas. Desafortunadamente, es más difícil resolver estos problemas a tiempo de ejecución a la vez que se mantiene un buen rendimiento a través de una amplia variedad de arquitecturas.

Soporte del Lenguaje para Mejorar el Rendimiento - Notas Finales

En este capítulo, hemos cubierto algunos de los esfuerzos que se han desarrollado en el área de los lenguajes para permitir la escritura de programas para cómputo escalable. Existe una pugna entre el FORTRAN-77 puro, el FORTRAN 90, el HPF y el paso de mensajes por convertirse en la herramienta definitiva para el cómputo escalable de alto rendimiento.

Ciertamente, ha habido ejemplos de grandes éxitos tanto de FORTRAN 90 (Thinking Machines CM-5) y HPF (IBM SP y otros) como lenguajes que pueden hacer un excelente uso de los sistemas de cómputo escalables. Uno de los problemas del enfoque basado en lenguajes de alto nivel es que, en algunas ocasiones, usar un lenguaje de alto nivel abstracto en realidad *reduce* la transportabilidad efectiva.

Los lenguajes están diseñados para ser transportables, pero si el vendedor de su computadora escalable particular no suporte la variante de lenguaje en la que usted ha elegido escribir su aplicación, entonces no será transportable. Incluso si el vendedor tiene disponible su lenguaje, puede que no esté afinado para generar el mejor código para su arquitectura.

Una solución estriba en comprar sus compiladores de una tercer compañía, tal como Pacific Sierra o Kuck y Asociados. Estos vendedores comercializan un compilador que se ejecuto sobre un amplio rango de sistema. Para aquellos usuarios que puedan costear tales opciones, estos compiladores proporcionan un mayor nivel de transportabilidad.

Una de las preocupaciones fundamentales es el problema del huevo y la gallina. Si los usuarios no emplean un lenguaje, los vendedores no le harán mejoras. Si todos los usuarios influyentes (con todo el dinero) usan paso de mensajes, entonces la existencia de un compilador excelente de HPF no tiene ningún valor real para esos usuarios.

Las buenas noticias son que tanto FORTRAN 90 como HPF proporcionan un camino hacia el cómputo escalable transportable que no requiere paso de mensajes explícito. La única pregunta es qué camino decidirán usar los usuarios.

Ambientes de Paso de Mensajes - Introducción

Una interfaz de paso de mensajes es un conjunto de funciones y llamados a subrutinas para C o FORTRAN, que le proporcionan una forma de dividir una aplicación para que pueda ejecutarse en paralelo. Los datos se dividen y pasan a otros procesadores en forma de mensajes. Los procesadores receptores los desempacan, realizan algún trabajo, y envían los resultados de vuelta, o bien los pasan a otros procesadores en la computadora paralela.

En cierta forma, el paso de mensajes es el "lenguaje ensamblador" del procesamiento paralelo. Carga sobre sus hombros la responsabilidad última, y si usted es talentoso (y el problema que trata de resolver coopera), le permite lograr el máximo rendimiento. Si tiene usted un problema fácilmente escalable y no está satisfecho con el rendimiento obtenido, muy probablemente sea su culpa, puesto que el compilador no sabe nada de los aspectos paralelos del programa.

Los dos ambientes de paso de mensajes más populares son *parallel virtual machine* (PVM) y *message-passing interface* (MPI). Muchas de las características importantes están disponibles en ambos ambientes. Una vez que se haya convertido en un experto en paso de mensajes, migrar de PVM a MPI no le dará problemas. Incluso podrá usted operar sobre un sistema que proporcione sólo una interfaz de paso de mensajes propietaria. Sin embargo, una vez que entienda los conceptos de paso de mensajes y haya descompuesto apropiadamente su aplicación, usualmente no le costará mucho más migrarlo de una biblioteca de paso de mensajes a otra.[footnote] Note que dije "no le costará mucho más.”

Ambientes de Paso de Mensajes - Parallel Virtual Machine

La idea tras PVM consiste en ensamblar una "máquina virtual" mediante un conjunto diverso de recursos enlazados por una red. Un usuario puede controlar los recursos de 35 estaciones disponibles sobre la Internet, y tener su propio sistema de procesamiento escalable. El trabajo de PVM comenzó a inicios de la década de 1990 en Oak Ridge National Labs, y en buena medida se convirtió en un éxito instantáneo en la comunidad científica. Proporciona un marco de trabajo en bruto que permite experimentar con el uso de redes de estaciones de trabajo como procesadores paralelos.

En PVM versión 3, puede usted crear su máquina virtual usando procesadores individuales, multiprocesadores de memoria compartida y multiprocesadores escalables. PVM intenta entretejer todos esos recursos en un ambiente de ejecución único y consistente.

Para ejecutar PVM, lo único que requiere usted es una cuenta de acceso en un conjunto de computadoras en red, que tengan instalado el software PVM. Incluso puede usted instalarlo en su propio directorio personal. Para crear su propia máquina virtual personal, debe integrar una lista de tales computadoras en un archivo:

```
% cat hostfile
frodo.egr.msu.edu
gollum.egr.msu.edu
mordor.egr.msu.edu
%
```

Tras ciertas manipulaciones no triviales de rutas de acceso y variables de ambiente, puede usted arrancar la consola PVM:

```
% pvm hostfile
pvmd already running.
```

```
      pvm> conf
      1 host, 1 data format
                          HOST     DTID     ARCH
SPEED
                        frodo     40000 SUN4SOL2
1000
                        gollum    40001 SUN4SOL2
1000
                        mordor    40002 SUN4SOL2
1000
      pvm> ps
                         HOST     TID   FLAG 0x
COMMAND
                          frodo    40042     6/c,f
pvmgs
      pvm> reset
      pvm> ps
                         HOST     TID   FLAG 0x
COMMAND
      pvm>
```

Muchos usuarios distintos puede ejecutar máquinas virtuales que usan el mismo grupo de recursos. Y cada usuario a ver como una máquina vacía. La única forma de que detecte usted las otras máquinas virtuales que emplean los mismos recursos que la suya, es mediante el porcentaje de tiempo que sus aplicaciones tendrán el control de la CPU.

Existen muchos comandos que pueden ejecutarse en la consola PVM. El comando *ps* muestra los procesos en ejecución en su máquina virtual. Es factible tener más procesos que sistemas de cómputo. Cada proceso comparte el tiempo de ejecución del sistema con el resto de la carga de trabajo del mismo. El comando *reset* sirve para reiniciar su máquina virtual. Usted es el administrador de sistemas virtual de la máquina virtual que ensambló.

Para poder ejecutar programas en su máquina virtual, debe usted compilar y enlazar sus programas con las rutinas de la biblioteca PVM: [footnote]
Nota: la forma exacta de compilar puede ser diferente para su sistema.

```
      % aimk mast slav
      making in SUN4SOL2/ for SUN4SOL2
      cc -O -I/opt/pvm3/include -DSYSVBFUNC -
DSYSVSTR -DNOGETDTBLSIZ
            -DSYSVSIGNAL -DNOWAIT3 -DNOUNIXDOM -o
mast
            ../mast.c -L/opt/pvm3/lib/SUN4SOL2 -
lpvm3 -lnsl -lsocket
      mv mast ~crs/pvm3/bin/SUN4SOL2
      cc -O -I/opt/pvm3/include -DSYSVBFUNC -
DSYSVSTR -DNOGETDTBLSIZ
            -DSYSVSIGNAL -DNOWAIT3 -DNOUNIXDOM -o
slav
            ../slav.c -L/opt/pvm3/lib/SUN4SOL2 -
lpvm3 -lnsl -lsocket
      mv slav ~crs/pvm3/bin/SUN4SOL2
      %
```

Cuando se topa con la primera llamada a PVM, la aplicación contacta con su máquina virtual y se registra a sí misma en ella. En este punto debe mostrarse como parte de la salida del comando *ps* si se ejecuta en la consola de la máquina virtual.

A partir de ese punto, su aplicación realiza llamadas a PVM para crear más procesos e interactuar con ellos. PVM tiene la responsabilidad de distribuir los procesos entre los diferentes sistemas en la máquina virtual, basado en la carga y su evaluación del rendimiento relativo de cada sistema. Los mensajes se mueven a lo largo de la red usando el *user datagram protocol* (UDP), y así se entregan al proceso apropiado.

Típicamente, la aplicación PVM arranca algunos procesos PVM adicionales. Puede tratarse de copias adicionales del mismo programa, o cada proceso PVM puede ejecutar una aplicación PVM diferente. Entonces el trabajo se distribuye entre los procesos, y los resultados se reúnen cuando sea necesario.

Existen varios modelos de cómputo básicos que pueden usarse típicamente cuando se trabaja con PVM:

- **Maestro/Esclavo** Cuando se opera en este modo, se designa un proceso (usualmente el inicial) como el maestro, que engendra cierto número de procesos que realizarán el trabajo. Se le envían unidades de trabajo a cada uno de estos procesos, y los resultados se regresan al maestro. A menudo el proceso maestro mantiene una cola de trabajos a realizarse, de forma que cuando el esclavo finaliza, el maestro le envía a éste un nuevo elemento de trabajo. Este enfoque funciona bien cuando hay poca interacción entre datos, es decir, cuando cada unidad de trabajo es independiente, y tiene la ventaja de que el problema global balancea su carga de manera natural, incluso cuando hay variaciones en los tiempos de ejecución de los procesos individuales.
- **Difusión/Reunión** Este tipo de aplicación se caracteriza típicamente por el hecho de que la estructura de datos compartida es relativamente pequeña, y puede copiarse fácilmente en cada nodo de procesamiento. Al inicio de cada ciclo, todas las estructuras de datos globales se envían por difusión del proceso maestro a todos los demás. Luego cada proceso opera sobre su propia porción de los datos, y cada uno produce un resultado parcial, que es enviado de vuelta para que el proceso maestro los reúna. Este patrón se repite ciclo tras ciclo.
- **SPMD/Descomposición de datos** Cuando la estructura de datos es demasiado grande como para que cada proceso almacena su propia copia, ésta debe dividirse entre múltiples procesos. Generalmente, al inicio de cada ciclo, todos los procesos deben intercambiar algunos datos con cada uno de sus procesos vecinos. Después, con sus datos locales acrecentados por el necesario subconjunto de datos remotos, realizan sus cáculos. Al final del ciclo, se intercambian nuevamente los datos necesarios entre los procesos vecinos, y se reinicia el ciclo.

Las aplicaciones mas complicadas tienen flujos de datos no uniformes, así como datos que migran entre los sistemas conforme la aplicación y las cargas de trabajo cambian en el sistema.

En esta sección tenemos dos programas de ejemplo: uno es una operación maestro-esclavo, y la otra es una solución del tipo descomposición de datos, ambos para el problema del flujo de calor.

Cola de Tareas

En este ejemplo, un proceso (`mast`) crea cuatro procesos esclavos (`slav`) y reparte 20 unidades de trabajo (sumar uno a un número). Conforme responde un proceso esclavo, se le da un nuevo trabajo o se le dice que se han agotado todas las unidades de trabajo:

```
% cat mast.c
#include <stdio.h>
#include "pvm3.h"

#define MAXPROC 5
#define JOBS 20

main()
{
  int mytid,info;
  int tids[MAXPROC];
  int tid,input,output,answers,work;

  mytid = pvm_mytid();
  info=pvm_spawn("slav", (char**)0, 0, "",
MAXPROC, tids);

/* Envía el primer trabajo */
  for(work=0;work<MAXPROC;work++) {
    pvm_initsend(PvmDataDefault);
```

```
          pvm_pkint(&work, 1, 1 ) ;
          pvm_send(tids[work],1) ;/* 1 = msgtype
*/
        }

      /* Envía el resto de las solicitudes de
trabajo */
        work = MAXPROC;
        for(answers=0; answers < JOBS ; answers++)
{
          pvm_recv( -1, 2 ); /* -1 = any task 2 =
msgtype */
          pvm_upkint( &tid, 1, 1 );
          pvm_upkint( &input, 1, 1 );
          pvm_upkint( &output, 1, 1 );
          printf("Gracias a %d
2*%d=%d\n",tid,input,output);
          pvm_initsend(PvmDataDefault);
          if ( work < JOBS ) {
             pvm_pkint(&work, 1, 1 ) ;
             work++;
          } else {
            input = -1;
            pvm_pkint(&input, 1, 1 ) ; /* Les
indica que se detengan */
          }
          pvm_send(tid,1) ;
        }

        pvm_exit();
      }
      %
```

Uno de los aspectos interesantes de la interfaz PVM es la separación de llamadas para preparar un nuevo mensaje, empaquetar datos en él y enviarlo. Ello es así por varias razones. PVM tiene la capacidad de

convertir entre distintos formatos de punto flotante, cambiar el orden de los bytes y traducir formatos de caracteres. Esto también permite que un único mensaje tenga múltiples elementos de datos con diferentes tipos.

El propósito del tipo de mensaje en cada envío o recepción PVM es permitir a quien envía esperar por un tipo particular de mensaje. En este ejemplo, usamos dos tipos de mensajes: el primero es un mensaje del maestro al esclavo, y el segundo es la respuesta.

Al realizar una recepción, un proceso puede o bien esperar un mensaje proveniente de un proceso específico, o un mensaje de cualquier proceso.

En la segunda fase del cómputo, el maestro espera la respuesta de cualquier esclavo, la imprime y luego reparte otra unidad de trabajo al esclavo, o le indica que termine, enviándole un mensaje con el valor -1.

El código del esclavo es muy simple: espera un mensaje, lo desempaqueta, comprueba si se trata de un mensaje de terminación, regresa una respuesta, y repite:

```
      % cat slav.c
      #include <stdio.h>
      #include "pvm3.h"

      /* Un programa sencillo para duplicar
enteros */
      main()
      {
        int mytid;
        int input,output;
        mytid = pvm_mytid();

        while(1) {
          pvm_recv( -1, 1 ); /* -1 = cualquier
tarea 1= tipo de mensaje */
          pvm_upkint(&input, 1, 1);
```

```
          if ( input == -1 ) break; /* Todo
realizado */

          output = input * 2;
          pvm_initsend( PvmDataDefault );
          pvm_pkint( &mytid, 1, 1 );
          pvm_pkint( &input, 1, 1 );
          pvm_pkint( &output, 1, 1 );
          pvm_send( pvm_parent(), 2 );
        }
        pvm_exit();
      }
      %
```

Cuando se ejecuta el programa maestro, produce la siguiente salida:

```
% pheat
Gracias a 262204 2*0=0
Gracias a 262205 2*1=2
Gracias a 262206 2*2=4
Gracias a 262207 2*3=6
Gracias a 262204 2*5=10
Gracias a 262205 2*6=12
Gracias a 262206 2*7=14
Gracias a 262207 2*8=16
Gracias a 262204 2*9=18
Gracias a 262205 2*10=20
Gracias a 262206 2*11=22
Gracias a 262207 2*12=24
Gracias a 262205 2*14=28
Gracias a 262207 2*16=32
Gracias a 262205 2*17=34
Gracias a 262207 2*18=36
Gracias a 262204 2*13=26
```

```
Gracias a 262205 2*19=38
Gracias a 262206 2*15=30
Gracias a 262208 2*4=8
%
```

Claramente los procesos están operando en paralelo, y el orden de ejecución es en cierta forma eleatorio. Este código es un excelente esqueleto para manejar una amplia variedad de cómputos. En el siguiente ejemplo, realizaremos un cómputo estilo SPMD para resolver el problema del flujo de calor usando PVM.

Flujo de Calor en PVM

El siguiente ejemplo es una aplicación mucho más complicada, que implementa el problema del flujo de calor en PVM. De muchos modos, nos da una vsión del trabajo que realiza el ambiente HPF. Resolveremos un flujo de calor en una placa bidimensional con dos fuentes de calor y los bordes inmersos en agua a cero grados, como se muestra en [link].

Una placa bidimensional con cuatro fuentes constantes de calor

Los datos se repartirán entre todos los procesos usando una distribución (`*`, `BLOCK`). Las columnas se distribuyen entre procesos en bloques contiguos, y todos los elementos de los renglones en una columna se almacenan en el mismo proceso. Como con HPF, el proceso que "posee" una celda de datos realiza los cálculos para esa celda tras recibir cualquier dato necesario para realizar el cálculo.

Usamos un enfoque rojo-negro, pero por simplicidad copiamos los datos de vuelta al final de cada iteración. Para que verdaderamente fuera rojo-negro,

debe usted realizar un cálculo en la dirección opuesta cada nuevo paso.

Note que en vez de engendrar procesos esclavos, el proceso padre engendra copias adicionales de sí mismo. Esto es típico de los programas estilo SPMD. Una vez engendrados tales procesos adicionales, todos los procesos esperan en una barrera antes de observar los números de proceso de los miembros del grupo. Una vez que los procesos han llegado a la barrera, todos ellos recuprean una lista de los distintos números de proceso:

```
      % cat pheat.f
            PROGRAM PHEAT
            INCLUDE '../include/fpvm3.h'
            INTEGER NPROC,ROWS,COLS,TOTCOLS,OFFSET
            PARAMETER(NPROC=4,MAXTIME=200)
            PARAMETER(ROWS=200,TOTCOLS=200)
            PARAMETER(COLS=(TOTCOLS/NPROC)+3)
            REAL*8 RED(0:ROWS+1,0:COLS+1),
BLACK(0:ROWS+1,0:COLS+1)
            LOGICAL IAMFIRST,IAMLAST
            INTEGER INUM,INFO,TIDS(0:NPROC-1),IERR
            INTEGER I,R,C
            INTEGER TICK,MAXTIME
            CHARACTER*30 FNAME

      *     Obtener como va la cosa SPMD - Unirse
al grupo pheat
            CALL PVMFJOINGROUP('pheat', INUM)

      * Si somos los primeros en el grupo pheat,
creamos algunos ayudantes
            IF ( INUM.EQ.0 ) THEN
              DO I=1,NPROC-1
                CALL PVMFSPAWN('pheat', 0,
'anywhere', 1, TIDS(I), IERR)
              ENDDO
            ENDIF
```

```
      *     Barrera para asegurarnos que todos
estamos en este punto, y así poder buscarnos
            CALL PVMFBARRIER( 'pheat', NPROC, INFO
)

      * Encontrar a mis camaradas y obtener sus
TIDs - Los TIDS son necesarios para los envíos
            DO I=0,NPROC-1
              CALL PVMFGETTID('pheat', I, TIDS(I))
            ENDDO
```

En este punto del código, tenemos `NPROC` procesos ejecutándose en modo `SPMD`. El siguiente paso es determinar cuál subconjunto del arreglo debe calcular cada proceso, lo cual se maneja mediante la variable `INUM`, cuyo rango va de 0 a 3 y que identifica unívocamente esos procesos.

Descomponemos los datos y almacenamos sólo un cuarto de los mismos en cada proceso. Usando la variable `INUM`, elegimos nuestro conjunto continuo de columnas para almacenar y calcular. La variable `OFFSET` mapea entre una columna "global" en el arreglo completo, y una columna local en su propio subconjunto del arreglo. [link] muestra un mapa que indica cuáles procesadores almacenan cuáles elementos de datos. Los valores marcados con una B son los valores de frontera, y no cambian durante la simulación. Todos ellos se ponen a 0. Este código a menudo resulta difícil de aprehender. Realizar una distribución (`BLOCK`, `BLOCK`) requiere una descomposición bidimensional e intercambiar datos con los vecinos superior e inferior, además de los vecinos izquierdo y derecho:
Asignación de los elementos de la cuadrícula a los procesadores

	0	1		50	51		100	101		150	151		200	201
0	B	B	B ...	B	B	B	B ... B	B	B	B ... B	B	B	B ... B	B	B
1	B	0	0 ...	0	0	1	1 ... 1	1	2	2 ... 2	2	3	3 ... 3	3	B
	B	0	0 ...	0	0	1	1 ... 1	1	2	2 ... 2	2	3	3 ... 3	3	B
...	B	0	0 ...	0	0	1	1 ... 1	1	2	2 ... 2	2	3	3 ... 3	3	B
	B	0	0 ...	0	0	1	1 ... 1	1	2	2 ... 2	2	3	3 ... 3	3	B
200	B	0	0 ...	0	0	1	1 ... 1	1	2	2 ... 2	2	3	3 ... 3	3	B
201	B	B	B ...	B	B	B	B ... B	B	B	B ... B	B	B	B ... B	B	B

```
      * Calcula mi geometría - ¿Qué subconjunto
debo procesar? (INUM=0 valores)
      * Columna actual = OFFSET + Columna (OFFSET
= 0)
      *     Columna 0 = vecinos a la izquierda
      *     Columna 1 = enviar a la izquierda
      *     Columnas 1..mylen Las celdas que me
toca calcular
      *     Columna mylen = Enviar a la derecha
(mylen=50)
      *     Column mylen+1 = Vecinos de la derecha
(Columna 51)

            IAMFIRST = (INUM .EQ. 0)
            IAMLAST = (INUM .EQ. NPROC-1)
            OFFSET = (ROWS/NPROC * INUM )
            MYLEN = ROWS/NPROC
            IF ( IAMLAST ) MYLEN = TOTCOLS -
OFFSET
            PRINT *,'INUM:',INUM,' Local',1,MYLEN,
           +                    '
Global',OFFSET+1,OFFSET+MYLEN

      * Inicio en frío
```

```
            DO C=0,COLS+1
              DO R=0,ROWS+1
                BLACK(R,C) = 0.0
              ENDDO
            ENDDO
```

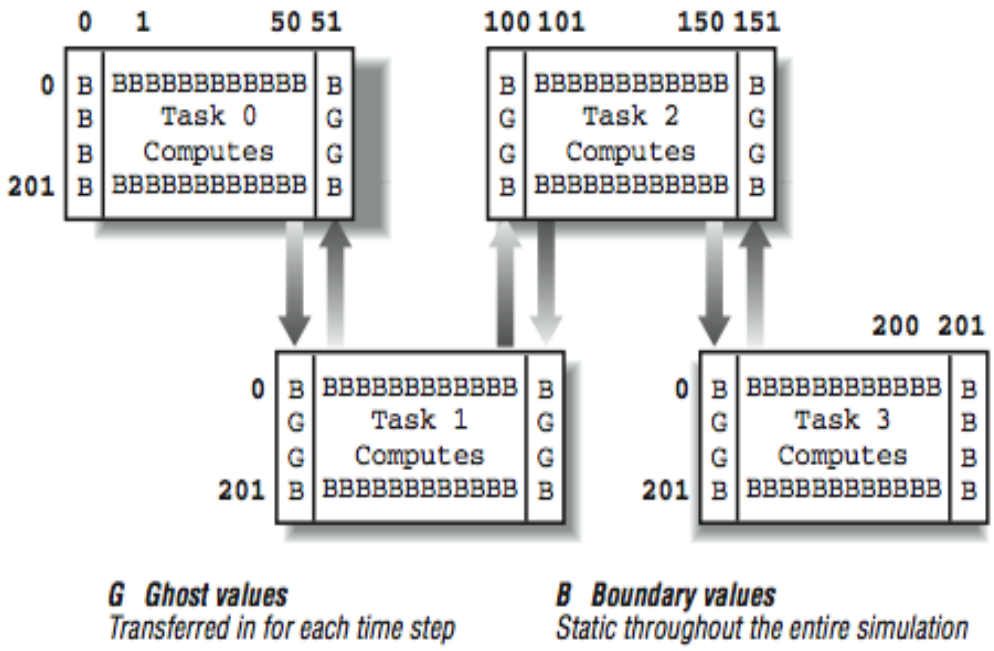
Ahora estamos ejecutando los pasos consecutivos. La primera acción en cada paso es reiniciar las fuentes de calor. En esta simulación, tenemos cuatro fuentes de calor colocadas cerca del centro de la placa. Debemos reiniciar todos los valores cada vez que ejecutamos la simulación, conforme se ven modificados por el ciclo principal:

```
      * Comenzamos la ejecución de los pasos
sucesivos
            DO TICK=1,MAXTIME

      * Configuramos las fuentes persistentes de
calor
              CALL
STORE(BLACK,ROWS,COLS,OFFSET,MYLEN,
           +     ROWS/3,TOTCOLS/3,10.0,INUM)
              CALL
STORE(BLACK,ROWS,COLS,OFFSET,MYLEN,
           +     2*ROWS/3,TOTCOLS/3,20.0,INUM)
              CALL
STORE(BLACK,ROWS,COLS,OFFSET,MYLEN,
           +     ROWS/3,2*TOTCOLS/3,-20.0,INUM)
              CALL
STORE(BLACK,ROWS,COLS,OFFSET,MYLEN,
           +     2*ROWS/3,2*TOTCOLS/3,20.0,INUM)
```

Ahora realizamos el intercambio de los "valores fantasmas" con nuestros procesos vecinos. Por ejemplo, el Proceso 0 contiene los elementos para la

columna global 50. Para calcular los valores para la columna 50 del paso siguiente , necesitamos la columna 51, que está almacenada en el Proceso 1. Similarmente, antes de que el Proceso 1 pueda calcular los nuevos valores para la columna 51, requiere los valores del Proceso 0 para la columna 50.

La [link] muestra cómo se transfieren los datos entre procesadores. Cada proceso envía su columna más a la izquierda al proceso a su izquierda, y su columna más a la derecha al proceso a su derecha. Dado que el primer y último procesos están rodeados por valores de frontera que no cambian a la izquierda y derecha, respectivamente, ello no es necesario para las columnas 1 y 200. Si todo se hace apropiadamente, cada proceso puede recibir sus valores fantasmas de sus vecinos izquierdo y derecho.
Patrón de comunicación de los valores fantasma



El resultado neto de todas las transferencias es que para cada espacio que debe calcularse, está rodeado por una capa ya sea de valores de frontera o valores fantasma de los vecinos izquierdo y derecho:

```
      * Enviar izquierdo y derecho
              IF ( .NOT. IAMFIRST ) THEN
                CALL PVMFINITSEND(PVMDEFAULT,TRUE)
                CALL PVMFPACK( REAL8, BLACK(1,1),
ROWS, 1, INFO )
                CALL PVMFSEND( TIDS(INUM-1), 1,
INFO )
              ENDIF
              IF ( .NOT. IAMLAST ) THEN
                CALL PVMFINITSEND(PVMDEFAULT,TRUE)
                CALL PVMFPACK( REAL8,
BLACK(1,MYLEN), ROWS, 1, INFO )
                CALL PVMFSEND( TIDS(INUM+1), 2,
INFO )
              ENDIF
      * Recibir derecho, luego izquierdo
              IF ( .NOT. IAMLAST ) THEN
                 CALL PVMFRECV( TIDS(INUM+1), 1,
BUFID )
                 CALL PVMFUNPACK ( REAL8,
BLACK(1,MYLEN+1), ROWS, 1, INFO
              ENDIF
              IF ( .NOT. IAMFIRST ) THEN
                CALL PVMFRECV( TIDS(INUM-1), 2,
BUFID )
                CALL PVMFUNPACK ( REAL8,
BLACK(1,0), ROWS, 1, INFO)
              ENDIF
```

El siguiente segmento es la parte fácil. Todos los valores fantasma apropiados están en su lugar, así que simplemente debemos realizar el cálculo de nuestro subespacio. Al final, copiamos de vuelta desde el arreglo `ROJO` al arreglo `NEGRO`; en una simulación real, podemos realizar dos pasos consecutivos, uno desde `NEGRO` hacia `ROJO` y el otro de `ROJO` hacia `NEGRO`, para ahorrar esta copia extra:

```
      * Realiza el flujo
              DO C=1,MYLEN
                DO R=1,ROWS
                  RED(R,C) = ( BLACK(R,C) +
           +                   BLACK(R,C-1) +
BLACK(R-1,C) +
           +                   BLACK(R+1,C) +
BLACK(R,C+1) ) / 5.0
                ENDDO
              ENDDO

      * Copia de vuelta - Normalmente haríamos una
versión roja y una negra de este ciclo
              DO C=1,MYLEN
                DO R=1,ROWS
                  BLACK(R,C) = RED(R,C)
                ENDDO
              ENDDO
            ENDDO
```

Ahora encontramos la celda central y la enviamos al proceso maestro (si es necesario) de forma que pueda imprimirse. También volcamos los datos en archivos para depuración o visualización posterior de los resultados. Cada archivo se hace único agregando a su nombre el número de instancia. Después el programa termina:

```
            CALL
SENDCELL(RED,ROWS,COLS,OFFSET,MYLEN,INUM,TIDS(0),
           +       ROWS/2,TOTCOLS/2)

      * Volcado de los datos para verificación
            IF ( ROWS .LE. 20 ) THEN
```

```
              FNAME = '/tmp/pheatout.' //
CHAR(ICHAR('0')+INUM)

OPEN(UNIT=9,NAME=FNAME,FORM='formatted')
              DO C=1,MYLEN
                WRITE(9,100)(BLACK(R,C),R=1,ROWS)
      100        FORMAT(20F12.6)
              ENDDO
             CLOSE(UNIT=9)
           ENDIF
      * Ponemos todo junto
            CALL PVMFBARRIER( 'pheat', NPROC, INFO
)
            CALL PVMFEXIT( INFO )

            END
```

La rutina `SENDCELL` encuentra una celda en particular, y la imprime en el proceso maestro. Esta rutina se invoca en un estilo `SPMD`: todos los procesos entran en ella, si bien no necesariamente al mismo tiempo. Dependiendo del `INUM` y de la celda que estemos observando, cada proceso puede hacer algo diferente.

Si la celda en cuestión está en el proceso maestro, y nosotros somos el proceso maestro, imprimimos. Ninguno de los demás procesos hace algo. Si la celda en cuestión está almacenada en otro proceso, aquél que la contenga la envía al proceso maestro. El proceso maestro recibe el valor y lo imprime. El resto de los procesos no hacen nada.

Se tata de un ejemplo simple de un código con un estilo típicamente SPMD. Todos los procesos ejecutan el codigo aproximadamente al mismo tiempo, pero en base a la información local de cada proceso, las acciones realizadas por los diferentes procesos pueden ser muy distintas:

```
SUBROUTINE
SENDCELL(RED,ROWS,COLS,OFFSET,MYLEN,INUM,PTID,R,C)
            INCLUDE '../include/fpvm3.h'
            INTEGER
ROWS,COLS,OFFSET,MYLEN,INUM,PTID,R,C
            REAL*8 RED(0:ROWS+1,0:COLS+1)
            REAL*8 CENTER

      * Calculamos el número de renglón local,
para determinar si es nuestro
            I = C - OFFSET
            IF ( I .GE. 1 .AND. I.LE. MYLEN ) THEN
              IF ( INUM .EQ. 0 ) THEN
                PRINT *,'Master has', RED(R,I), R,
C, I
              ELSE
                CALL PVMFINITSEND(PVMDEFAULT,TRUE)
                CALL PVMFPACK( REAL8, RED(R,I), 1,
1, INFO )
                PRINT *, 'INUM:',INUM,'
Returning',R,C,RED(R,I),I
                CALL PVMFSEND( PTID, 3, INFO )
              ENDIF
            ELSE
              IF ( INUM .EQ. 0 ) THEN
                CALL PVMFRECV( -1 , 3, BUFID )
                CALL PVMFUNPACK ( REAL8, CENTER,
1, 1, INFO)
                PRINT *, 'Master
Received',R,C,CENTER
              ENDIF
            ENDIF
            RETURN
            END
```

Como la rutina previa, todos los procesos ejecutan la rutina `STORE`. La idea es almacenar un valor en una posición de renglón y columna *global*. Primero, debemos determinar si la celda está en nuestro proceso. Si es así, debemos calcular la columna local (I) en nuestro subconjunto de la matriz global, y luego almacenar el valor:

```
      SUBROUTINE
STORE(RED,ROWS,COLS,OFFSET,MYLEN,R,C,VALUE,INUM)
      REAL*8 RED(0:ROWS+1,0:COLS+1)
      REAL VALUE
      INTEGER ROWS,COLS,OFFSET,MYLEN,R,C,I,INUM
      I = C - OFFSET
      IF ( I .LT. 1 .OR. I .GT. MYLEN ) RETURN
      RED(R,I) = VALUE
      RETURN
      END
```

Cuando se ejecuta este programa, proporciona la siguiente salida:

```
      % pheat
       INUM: 0 Local 1 50 Global 1 50
       El maestro recibió 100 100
3.4722390023541D-07
      %
```

Vemos do líneas de impresión. La primera indica los valores que el Proceso 0 usó en su cálculo de geometría. La segunda es la salida del proceso maestro con la temperatura de la celda (100,100), tras 200 ciclos de tiempo.

Una técnica interesante, que resulta útil para depurar este tipo de programa, es cambiar el número de procesos creados. Si el programa no está moviendo sus datos apropiadamente, usualmente obtendrá usted distintos resultados

cuando use distinto número de procesos. Si lo observa detalladamente, notará que el código anterior funciona correctamente de 1 a 30 procesos.

Observe que no hay una operación de barrera al final de cada ciclo de tiempo. Ello contrasta con la forma en que operan los ciclos paralelos en los multiprocesadores con acceso a uniforme a memoria compartida, que fuerzan una barrera al final de cada ciclo. Dado que hemos puesto como regla que "el dueño calcule", y que nada se calcule hasta que se hayan recibido todos los datos fantasma, no hay necesidad de tal barrera. El receptor de los mensajes con los valores fantasmas apropiados permite a un proceso comenzar a calcular de inmediato, sin importar lo que los otros procesos estén realizando en ese momento.

Este ejemplo puede usarse ya sea como un marco de trabajo para desarrollar otros cálculos basados en retículas, o como una buena excusa para usar HPF y apreciar el arduo trabajo llevado a cabo por los creadores de compiladores HPF. Una implementación bien hecha de esta simulación en HPF debe presentar mejor rendimiento que la implementación PVM, porque HPF puede llevar a cabo optimizaciones más estrictas. Al contrario que nosotros, el compilador HPF no tiene por qué hacer que sea fácilmente legible el código que genera.

PVM Summary

PVM es una herramienta ampliamente usada, porque proporciona portabilidad a lo largo de cualquier arquitectura diferente de la SIMD. Una vez realizado el esfuerzo de hacer que un código use paso de mensajes, tiende a ejecutarse bien en muchas arquitecturas distintas.

Las principales preocupaciones cuando se usa PVM son:

- La necesidad de una etapa de empaquetamiento, separada de la etapa de envío.
- El hecho de que esté diseñado para trabajar en un ambiente heterogéneo, puede implicar cierto nivel de sobrecarga
- No lleva cabo de manera automática tareas comunes como los cálculos geométricos

Pero a pesar de todo, para cierto conjunto de programadores, PVM es la herramienta a usarse. Si desea usted aprender más acerca de PVM vea *PVM — A User's Guide and Tutorial for Networked Parallel Computing*, de Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, y Vaidy Sunderam (MIT Press). Hay información disponible acerca de ello en www.netlib.org/pvm3/.

Message-Passing Interface

The Message-Passing Interface (MPI) was designed to be an industrial-strength message-passing environment that is portable across a wide range of hardware environments.

Much like High Performance FORTRAN, MPI was developed by a group of computer vendors, application developers, and computer scientists. The idea was to come up with a specification that would take the strengths of many of the existing proprietary message passing environments on a wide variety of architectures and come up with a specification that could be implemented on architectures ranging from SIMD systems with thousands of small processors to MIMD networks of workstations and everything in between.

Interestingly, the MPI effort was completed a year *after* the High Performance FORTRAN (HPF) effort was completed. Some viewed MPI as a portable message-passing interface that could support a good HPF compiler. Having MPI makes the compiler more portable. Also having the compiler use MPI as its message-passing environment insures that MPI is heavily tested and that sufficient resources are invested into the MPI implementation.

PVM Versus MPI

While many of the folks involved in PVM participated in the MPI effort, MPI is not simply a follow-on to PVM. PVM was developed in a university/research lab environment and evolved over time as new features were needed. For example, the group capability was not designed into PVM at a fundamental level. Some of the underlying assumptions of PVM were based “on a network of workstations connected via Ethernet” model and didn’t export well to scalable computers.[footnote] In some ways, MPI is more robust than PVM, and in other ways, MPI is simpler than PVM. MPI doesn’t specify the system management details as in PVM; MPI doesn’t specify how a virtual machine is to be created, operated, and used.
One should not diminish the positive contributions of PVM, however. PVM was the first widely avail- able portable message-passing environment.

PVM pioneered the idea of heterogeneous distributed computing with built-in format conversion.

MPI Features

MPI has a number of useful features beyond the basic send and receive capabilities. These include:

- **Communicators**: A communicator is a subset of the active processes that can be treated as a group for collective operations such as broadcast, reduction, barriers, sending, or receiving. Within each communicator, a process has a *rank* that ranges from zero to the size of the group. A process may be a member of more than one communicator and have a different rank within each communicator. There is a default communicator that refers to all the MPI processes that is called `MPI_COMM_WORLD`.
- **Topologies**: A communicator can have a topology associated with it. This arranges the processes that belong to a communicator into some layout. The most common layout is a Cartesian decomposition. For example, 12 processes may be arranged into a 3×4 grid.[footnote] Once these topologies are defined, they can be queried to find the neighboring processes in the topology. In addition to the Cartesian (grid) topology, MPI also supports a graph-based topology.
  Sounds a little like HPF, no?
- **Communication modes**: MPI supports multiple styles of communication, including blocking and non- blocking. Users can also choose to use explicit buffers for sending or allow MPI to manage the buffers. The nonblocking capabilities allow the overlap of communication and computation. MPI can support a model in which there is no available memory space for buffers and the data must be copied directly from the address space of the sending process to the memory space of the receiving process. MPI also supports a single call to perform a send and receive that is quite useful when processes need to exchange data.
- **Single-call collective operations**: Some of the calls in MPI automate collective operations in a single call. For example, the broadcast operation sends values from the master to the slaves and receives the

values on the slaves in the same operation. The net result is that the values are updated on all processes. Similarly, there is a single call to sum a value across all of the processes to a single value. By bundling all this functionality into a single call, systems that have support for collective operations in hardware can make best use of this hardware. Also, when MPI is operating on a shared-memory environment, the broadcast can be simplified as all the slaves simply make a local copy of a shared variable.

Clearly, the developers of the MPI specification had significant experience with developing message-passing applications and added many widely used features to the message-passing library. Without these features, each programmer needed to use more primitive operations to construct their own versions of the higher-level operations.

Heat Flow in MPI

In this example, we implement our heat flow problem in MPI using a similar decomposition to the PVM example. There are several ways to approach the prob- lem. We could almost translate PVM calls to corresponding MPI calls using the `MPI_COMM_WORLD` communicator. However, to showcase some of the MPI features, we create a Cartesian communicator:

```
            PROGRAM MHEATC
            INCLUDE 'mpif.h'
            INCLUDE 'mpef.h'
            INTEGER ROWS,COLS,TOTCOLS
            PARAMETER(MAXTIME=200)
      * This simulation can be run on MINPROC or
greater processes.
      * It is OK to set MINPROC to 1 for testing
purposes
      * For a large number of rows and columns, it
is best to set MINPROC
```

```
      * to the actual number of runtime processes
            PARAMETER(MINPROC=2)
PARAMETER(ROWS=200,TOTCOLS=200,COLS=TOTCOLS/MINPRO
C)
            DOUBLE PRECISION
RED(0:ROWS+1,0:COLS+1),BLACK(0:ROWS+1,0:COLS+1)
            INTEGER S,E,MYLEN,R,C
            INTEGER TICK,MAXTIME
            CHARACTER*30 FNAME
```

The basic data structures are much the same as in the PVM example. We allocate a subset of the heat arrays in each process. In this example, the amount of space allocated in each process is set by the compile-time variable `MINPROC`. The simulation can execute on more than `MINPROC` processes (wasting some space in each process), but it can't execute on less than `MINPROC` processes, or there won't be sufficient total space across all of the processes to hold the array:

```
      INTEGER COMM1D,INUM,NPROC,IERR
      INTEGER DIMS(1),COORDS(1)
      LOGICAL PERIODS(1)
      LOGICAL REORDER
      INTEGER NDIM
      INTEGER STATUS(MPI_STATUS_SIZE)
      INTEGER RIGHTPROC, LEFTPROC
```

These data structures are used for our interaction with MPI. As we will be doing a one-dimensional Cartesian decomposition, our arrays are dimensioned to one. If you were to do a two-dimensional decomposition, these arrays would need two elements:

```
      PRINT *,'Calling MPI_INIT'
```

```
      CALL MPI_INIT( IERR )
      PRINT *,'Back from MPI_INIT'
      CALL MPI_COMM_SIZE( MPI_COMM_WORLD, NPROC,
IERR )
```

The call to `MPI_INIT` creates the appropriate number of processes. Note that in the output, the PRINT statement before the call only appears once, but the second PRINT appears once for each process. We call `MPI_COMM_SIZE` to determine the size of the global communicator `MPI_COMM_WORLD`. We use this value to set up our Cartesian topology:

```
      * Create new communicator that has a
Cartesian topology associated
      * with it - MPI_CART_CREATE returns COMM1D -
A communicator descriptor

            DIMS(1) = NPROC
            PERIODS(1) = .FALSE.
            REORDER = .TRUE.
            NDIM = 1

            CALL MPI_CART_CREATE(MPI_COMM_WORLD,
NDIM, DIMS, PERIODS,
            +     REORDER, COMM1D, IERR)
```

Now we create a one-dimensional (`NDIM=1`) arrangement of all of our processes (`MPI_COMM_WORLD`). All of the parameters on this call are input values except for `COMM1D` and `IERR`. `COMM1D` is an integer "communicator handle." If you print it out, it will be a value such as 134. It is not actually data, it is merely a handle that is used in other calls. It is quite similar to a file descriptor or unit number used when performing input-output to and from files.

The topology we use is a one-dimensional decomposition that isn't periodic. If we specified that we wanted a periodic decomposition, the far-left and far-right processes would be neighbors in a wrapped-around fashion making a ring. Given that it isn't periodic, the far-left and far-right processes have no neighbors.

In our PVM example above, we declared that Process 0 was the far-right process, Process `NPROC-1` was the far-left process, and the other processes were arranged linearly between those two. If we set `REORDER` to `.FALSE.`, `MPI` also chooses this arrangement. However, if we set `REORDER` to `.TRUE.`, `MPI` may choose to arrange the processes in some other fashion to achieve better performance, assuming that you are communicating with close neighbors.

Once the communicator is set up, we use it in all of our communication operations:

```
      * Get my rank in the new communicator

            CALL MPI_COMM_RANK( COMM1D, INUM,
IERR)
```

Within each communicator, each process has a rank from zero to the size of the communicator minus 1. The `MPI_COMM_RANK` tells each process its rank within the communicator. A process may have a different rank in the `COMM1D` communicator than in the `MPI_COMM_WORLD` communicator because of some reordering.

Given a Cartesian topology communicator,[footnote] we can extract information from the communicator using the `MPI_CART_GET` routine: Remember, each communicator may have a topology associated with it. A topology can be grid, graph, or none. Interestingly, the `MPI_COMM_WORLD` communicator has no topology associated with it.

```
      * Given a communicator handle COMM1D, get
the topology, and my position
      * in the topology

            CALL MPI_CART_GET(COMM1D, NDIM, DIMS,
PERIODS, COORDS, IERR)
```

In this call, all of the parameters are output values rather than input values as in the `MPI_CART_CREATE` call. The `COORDS` variable tells us our coordinates within the communicator. This is not so useful in our one-dimensional example, but in a two-dimensional process decomposition, it would tell our current position in that two-dimensional grid:

```
      * Returns the left and right neighbors 1
unit away in the zeroth dimension
      * of our Cartesian map - since we are not
periodic, our neighbors may
      * not always exist - MPI_CART_SHIFT handles
this for us

            CALL MPI_CART_SHIFT(COMM1D, 0, 1,
LEFTPROC, RIGHTPROC, IERR)
            CALL MPE_DECOMP1D(TOTCOLS, NPROC,
INUM, S, E)
            MYLEN = ( E - S ) + 1
            IF ( MYLEN.GT.COLS ) THEN
              PRINT *,'Not enough space,
need',MYLEN,' have ',COLS
              PRINT *,TOTCOLS,NPROC,INUM,S,E
              STOP
            ENDIF
            PRINT
```

```
*,INUM,NPROC,COORDS(1),LEFTPROC,RIGHTPROC, S, E
```

We can use `MPI_CART_SHIFT` to determine the rank number of our left and right neighbors, so we can exchange our common points with these neighbors. This is necessary because we can't simply send to `INUM-1` and `INUM+1` if MPI has chosen to reorder our Cartesian decomposition. If we are the far-left or far-right process, the neighbor that doesn't exist is set to `MPI_PROC_NULL`, which indicates that we have no neighbor. Later when we are performing message sending, it checks this value and sends messages only to real processes. By not sending the message to the "null process," MPI has saved us an `IF` test.

To determine which strip of the global array we store and compute in this process, we call a utility routine called `MPE_DECOMP1D` that simply does several calculations to evenly split our 200 columns among our processes in contiguous strips. In the PVM version, we need to perform this computation by hand.

The `MPE_DECOMP1D` routine is an example of an extended MPI library call (hence the MPE prefix). These extensions include graphics support and logging tools in addition to some general utilities. The MPE library consists of routines that were useful enough to standardize but not required to be supported by all MPI implementations. You will find the MPE routines supported on most MPI implementations.

Now that we have our communicator group set up, and we know which strip each process will handle, we begin the computation:

```
* Start Cold

      DO C=0,COLS+1
        DO R=0,ROWS+1
          BLACK(R,C) = 0.0
        ENDDO
```

```
            ENDDO
```

As in the PVM example, we set the plate (including boundary values) to zero.

All processes begin the time step loop. Interestingly, like in PVM, there is no need for any synchronization. The messages implicitly synchronize our loops.

The first step is to store the permanent heat sources. We need to use a routine because we must make the store operations relative to our strip of the global array:

```
      * Begin running the time steps
            DO TICK=1,MAXTIME

      * Set the persistent heat sources
              CALL
STORE(BLACK,ROWS,COLS,S,E,ROWS/3,TOTCOLS/3,10.0,IN
UM)
              CALL
STORE(BLACK,ROWS,COLS,S,E,2*ROWS/3,TOTCOLS/3,20.0,
INUM)
              CALL
STORE(BLACK,ROWS,COLS,S,E,ROWS/3,2*TOTCOLS/3,-20.0
,INUM)
              CALL
STORE(BLACK,ROWS,COLS,S,E,2*ROWS/3,2*TOTCOLS/3,20.
0,INUM)
```

All of the processes set these values independently depending on which process has which strip of the overall array.

Now we exchange the data with our neighbors as determined by the Cartesian communicator. Note that we don't need an IF test to determine if we are the far-left or far-right process. If we are at the edge, our neighbor setting is `MPI_PROC_NULL` and the `MPI_SEND` and `MPI_RECV` calls do nothing when given this as a source or destination value, thus saving us an IF test.

Note that we specify the communicator `COMM1D` because the rank values we are using in these calls are relative to that communicator:

```
      * Send left and receive right
              CALL
MPI_SEND(BLACK(1,1),ROWS,MPI_DOUBLE_PRECISION,
           +
LEFTPROC,1,COMM1D,IERR)
              CALL
MPI_RECV(BLACK(1,MYLEN+1),ROWS,MPI_DOUBLE_PRECISIO
N,
           +
RIGHTPROC,1,COMM1D,STATUS,IERR)

      * Send Right and Receive left in a single
statement
              CALL MPI_SENDRECV(
           +
BLACK(1,MYLEN),ROWS,COMM1D,RIGHTPROC,2,
           +
BLACK(1,0),ROWS,COMM1D,LEFTPROC, 2,
           +       MPI_COMM_WORLD, STATUS, IERR)
```

Just to show off, we use both the separate send and receive, and the combined send and receive. When given a choice, it's probably a good idea to use the combined operations to give the runtime environment more flexibility in terms of buffering. One downside to this that occurs on a

network of workstations (or any other high-latency interconnect) is that you can't do both send operations first and then do both receive operations to overlap some of the communication delay.

Once we have all of our ghost points from our neighbors, we can perform the algorithm on our subset of the space:

```
      * Perform the flow
              DO C=1,MYLEN
                DO R=1,ROWS
                  RED(R,C) = ( BLACK(R,C) +
           +                   BLACK(R,C-1) +
BLACK(R-1,C) +
           +                   BLACK(R+1,C) +
BLACK(R,C+1) ) / 5.0
                ENDDO
              ENDDO

      * Copy back - Normally we would do a red and
black version of the loop
            DO C=1,MYLEN
              DO R=1,ROWS
                BLACK(R,C) = RED(R,C)
              ENDDO
            ENDDO
          ENDDO
```

Again, for simplicity, we don't do the complete red-black computation. [footnote] We have no synchronization at the bottom of the loop because the messages implicitly synchronize the processes at the top of the next loop.

Note that you could do two time steps (one black-red-black iteration) if you exchanged two ghost columns at the top of the loop.

Again, we dump out the data for verification. As in the PVM example, one good test of basic correctness is to make sure you get exactly the same results for varying numbers of processes:

```
      * Dump out data for verification
            IF ( ROWS .LE. 20 ) THEN
              FNAME = '/tmp/mheatcout.' //
CHAR(ICHAR('0')+INUM)

OPEN(UNIT=9,NAME=FNAME,FORM='formatted')
              DO C=1,MYLEN
                WRITE(9,100)(BLACK(R,C),R=1,ROWS)
      100        FORMAT(20F12.6)
              ENDDO
             CLOSE(UNIT=9)
           ENDIF
```

To terminate the program, we call `MPI_FINALIZE`:

```
      * Lets all go together
            CALL MPI_FINALIZE(IERR)
            END
```

As in the PVM example, we need a routine to store a value into the proper strip of the global array. This routine simply checks to see if a particular global element is in this process and if so, computes the proper location within its strip for the value. If the global element is not in this process, this routine simply returns doing nothing:

```
            SUBROUTINE
```

```
STORE(RED,ROWS,COLS,S,E,R,C,VALUE,INUM)
            REAL*8 RED(0:ROWS+1,0:COLS+1)
            REAL VALUE
            INTEGER ROWS,COLS,S,E,R,C,I,INUM
            IF ( C .LT. S .OR. C .GT. E ) RETURN
            I = ( C - S ) + 1
      *     PRINT *,'STORE,
INUM,R,C,S,E,R,I',INUM,R,C,S,E,R,I,VALUE RED(R,I)
= VALUE
            RETURN
            END
```

When this program is executed, it has the following output:

```
      % mpif77 -c mheatc.f mheatc.f:
      MAIN mheatc:
      store:
      % mpif77 -o mheatc mheatc.o -lmpe
      % mheatc -np 4
      Calling MPI_INIT
      Back from MPI_INIT
      Back from MPI_INIT
      Back from MPI_INIT
      Back from MPI_INIT
      0  4  0  -1  1  1  50
      2  4  2  1  3  101  150
      3  4  3  2  -1  151  200
      1  4  1  0  2  51  100
     %
```

As you can see, we call `MPI_INIT` to activate the four processes. The `PRINT` statement immediately after the `MPI_INIT` call appears four times, once for each of the activated processes. Then each process prints out the

strip of the array it will process. We can also see the neighbors of each process including -1 when a process has no neighbor to the left or right. Notice that Process 0 has no left neighbor, and Process 3 has no right neighbor. MPI has provided us the utilities to simplify message-passing code that we need to add to implement this type of grid- based application.

When you compare this example with a PVM implementation of the same problem, you can see some of the contrasts between the two approaches. Programmers who wrote the same six lines of code over and over in PVM combined them into a single call in MPI. In MPI, you can think “data parallel” and express your program in a more data-parallel fashion.

In some ways, MPI feels less like assembly language than PVM. However, MPI does take a little getting used to when compared to PVM. The concept of a Cartesian communicator may seem foreign at first, but with understanding, it becomes a flexible and powerful tool.

Heat in MPI Using Broadcast/Gather

One style of parallel programming that we have not yet seen is the *broadcast/gather* style. Not all applications can be naturally solved using this style of programming. However, if an application can use this approach effectively, the amount of modification that is required to make a code run in a message-passing environment is minimal.

Applications that most benefit from this approach generally do a lot of computation using some small amount of shared information. One requirement is that one complete copy of the “shared” information must fit in each of the processes.

If we keep our grid size small enough, we can actually program our heat flow application using this approach. This is almost certainly a less efficient implementation than any of the earlier implementations of this problem because the core computation is so simple. However, if the core computations were more complex and needed access to values farther than one unit away, this might be a good approach.

The data structures are simpler for this approach and, actually, are no different than the single-process FORTRAN 90 or HPF versions. We will allocate a complete `RED` and `BLACK` array in every process:

```
      PROGRAM MHEAT
      INCLUDE 'mpif.h'
      INCLUDE 'mpef.h'
      INTEGER ROWS,COLS
      PARAMETER(MAXTIME=200)
      PARAMETER(ROWS=200,COLS=200)
      DOUBLE PRECISION
RED(0:ROWS+1,0:COLS+1),BLACK(0:ROWS+1,0:COLS+1)
```

We need fewer variables for the MPI calls because we aren't creating a communicator. We simply use the default communicator `MPI_COMM_WORLD`. We start up our processes, and find the size and rank of our process group:

```
      INTEGER INUM,NPROC,IERR,SRC,DEST,TAG
      INTEGER S,E,LS,LE,MYLEN
      INTEGER STATUS(MPI_STATUS_SIZE)
      INTEGER I,R,C
      INTEGER TICK,MAXTIME
      CHARACTER*30 FNAME

      PRINT *,'Calling MPI_INIT'
      CALL MPI_INIT( IERR )
      CALL MPI_COMM_SIZE( MPI_COMM_WORLD, NPROC,
IERR )
      CALL MPI_COMM_RANK( MPI_COMM_WORLD, INUM,
IERR)
      CALL MPE_DECOMP1D(COLS, NPROC, INUM, S, E,
```

```
IERR)
      PRINT *,'My Share ', INUM, NPROC, S, E
```

Since we are broadcasting initial values to all of the processes, we only have to set things up on the master process:

```
      * Start Cold

            IF ( INUM.EQ.0 ) THEN
              DO C=0,COLS+1
                DO R=0,ROWS+1
                  BLACK(R,C) = 0.0
                ENDDO
              ENDDO
            ENDIF
```

As we run the time steps (again with no synchronization), we set the persistent heat sources directly. Since the shape of the data structure is the same in the master and all other processes, we can use the real array coordinates rather than mapping them as with the previous examples. We could skip the persistent settings on the nonmaster processes, but it doesn't hurt to do it on all processes:

```
      * Begin running the time steps
            DO TICK=1,MAXTIME

      * Set the heat sources
              BLACK(ROWS/3, COLS/3)= 10.0
              BLACK(2*ROWS/3, COLS/3) = 20.0
              BLACK(ROWS/3, 2*COLS/3) = -20.0
              BLACK(2*ROWS/3, 2*COLS/3) = 20.0
```

Now we broadcast the entire array from process rank zero to all of the other processes in the `MPI_COMM_WORLD` communicator. Note that this call does the sending on rank zero process and receiving on the other processes. The net result of this call is that all the processes have the values formerly in the master process in a single call:

```
      * Broadcast the array
            CALL MPI_BCAST(BLACK,(ROWS+2)*
(COLS+2),MPI_DOUBLE_PRECISION,
           +
0,MPI_COMM_WORLD,IERR)
```

Now we perform the subset computation on each process. Note that we are using global coordinates because the array has the same shape on each of the processes. All we need to do is make sure we set up our particular strip of columns according to S and E:

```
      * Perform the flow on our subset

              DO C=S,E
                DO R=1,ROWS
                  RED(R,C) = ( BLACK(R,C) +
           +                   BLACK(R,C-1) +
BLACK(R-1,C) +
           +                   BLACK(R+1,C) +
BLACK(R,C+1) ) / 5.0
                ENDDO
              ENDDO
```

Now we need to gather the appropriate strips from the processes into the appropriate strip in the master array for rebroadcast in the next time step.

We could change the loop in the master to receive the messages in any order and check the `STATUS` variable to see which strip it received:

```
      * Gather back up into the BLACK array in
master (INUM = 0)
            IF ( INUM .EQ. 0 ) THEN
              DO C=S,E
                DO R=1,ROWS
                  BLACK(R,C) = RED(R,C)
                ENDDO
              ENDDO
            DO I=1,NPROC-1
              CALL MPE_DECOMP1D(COLS, NPROC, I,
LS, LE, IERR)
              MYLEN = ( LE - LS ) + 1
              SRC = I TAG = 0
              CALL MPI_RECV(BLACK(0,LS),MYLEN*
(ROWS+2),
           +
MPI_DOUBLE_PRECISION, SRC, TAG,
           +                     MPI_COMM_WORLD,
STATUS, IERR)
      *         Print *,'Recv',I,MYLEN
            ENDDO
           ELSE
            MYLEN = ( E - S ) + 1
            DEST = 0
            TAG = 0
            CALL MPI_SEND(RED(0,S),MYLEN*
(ROWS+2),MPI_DOUBLE_PRECISION,
           +                DEST, TAG,
MPI_COMM_WORLD, IERR)
                Print *,'Send',INUM,MYLEN
            ENDIF
```

```
            ENDDO
```

We use `MPE_DECOMP1D` to determine which strip we're receiving from each process.

In some applications, the value that must be gathered is a sum or another single value. To accomplish this, you can use one of the MPI reduction routines that coalesce a set of distributed values into a single value using a single call.

Again at the end, we dump out the data for testing. However, since it has all been gathered back onto the master process, we only need to dump it on one process:

```
      * Dump out data for verification
            IF ( INUM .EQ.0 .AND. ROWS .LE. 20 )
THEN
              FNAME = '/tmp/mheatout'

OPEN(UNIT=9,NAME=FNAME,FORM='formatted')
              DO C=1,COLS
                WRITE(9,100)(BLACK(R,C),R=1,ROWS)
      100        FORMAT(20F12.6)
              ENDDO
             CLOSE(UNIT=9)
            ENDIF

            CALL MPI_FINALIZE(IERR)
            END
```

When this program executes with four processes, it produces the following output:

```
% mpif77 -c mheat.f
mheat.f:
MAIN mheat:
% mpif77 -o mheat mheat.o -lmpe
% mheat -np 4
Calling MPI_INIT
My Share 1 4 51 100
My Share 0 4 1 50
My Share 3 4 151 200
My Share 2 4 101 150
%
```

The ranks of the processes and the subsets of the computations for each process are shown in the output.

So that is a somewhat contrived example of the broadcast/gather approach to parallelizing an application. If the data structures are the right size and the amount of computation relative to communication is appropriate, this can be a very effective approach that may require the smallest number of code modifications compared to a single-processor version of the code.

MPI Summary

Whether you chose PVM or MPI depends on which library the vendor of your system prefers. Sometimes MPI is the better choice because it contains the newest features, such as support for hardware-supported multicast or broadcast, that can significantly improve the overall performance of a scatter-gather application.

A good text on MPI is Using *MPI — Portable Parallel Programmingwith the Message-Passing Interface*, by William Gropp, Ewing Lusk, and Anthony Skjellum (MIT Press). You may also want to retrieve and print the MPI specification from http://www.netlib.org/mpi/.

Closing Notes

In this chapter we have looked at the “assembly language” of parallel programming. While it can seem daunting to rethink your application, there are often some simple changes you can make to port your code to message passing. Depending on the application, a master-slave, broadcast-gather, or decomposed data approach might be most appropriate.

It’s important to realize that some applications just don’t decompose into message passing very well. You may be working with just such an application. Once you have some experience with message passing, it becomes easier to identify the critical points where data must be communicated between processes.

While HPF, PVM, and MPI are all mature and popular technologies, it’s not clear whether any of these technologies will be the long-term solution that we will use 10 years from now. One possibility is that we will use FORTRAN 90 (or FORTRAN 95) without any data layout directives or that the directives will be optional. Another interesting possibility is simply to keep using FORTRAN 77. As scalable, cache-coherent, non-uniform memory systems become more popular, they will evolve their own data allocation primitives. For example, the HP/Exemplar supports the following data storage attributes: shared, node-private, and thread-private. As dynamic data structures are allocated, they can be placed in any one of these classes. Node-private memory is shared across all the threads on a single node but not shared beyond those threads. Perhaps we will only have to declare the storage class of the data but not the data layout for these new machines.

PVM and MPI still need the capability of supporting a fault-tolerant style of computing that allows an application to complete as resources fail or otherwise become available. The amount of compute power that will be available to applications that can tolerate some unreliability in the resources will be very large. There have been a number of moderately successful attempts in this area such as Condor, but none have really caught on in the mainstream.

To run the most powerful computers in the world at their absolute maximum performance levels, the need to be portable is somewhat reduced. Making your particular application go ever faster and scale to ever higher numbers of processors is a fascinating activity. May the FLOPS be with you!

Introduction

High Performance Microprocessors

It has been said that history is rewritten by the victors. It is clear that high performance RISC-based microprocessors are defining the current history of high performance computing. We begin our study with the basic building blocks of modern high performance computing: the high performance RISC microprocessors.

A *complex instruction set computer* (CISC) instruction set is made up of powerful primitives, close in functionality to the primitives of high-level languages like C or FORTRAN. It captures the sense of "don't do in software what you can do in hardware." RISC, on the other hand, emphasizes low-level primitives, far below the complexity of a high-level language. You can compute anything you want using either approach, though it will probably take more machine instructions if you're using RISC. The important difference is that with RISC you can trade instruction-set complexity for speed.

To be fair, RISC isn't really all that new. There were some important early machines that pioneered RISC philosophies, such as the CDC 6600 (1964) and the IBM 801 project (1975). It was in the mid-1980s, however, that RISC machines first posed a direct challenge to the CISC installed base. Heated debate broke out — RISC versus CISC — and even lingers today, though it is clear that the RISC[footnote] approach is in greatest favor; late-generation CISC machines are looking more RISC-like, and some very old families of CISC, such as the DEC VAX, are being retired.
One of the most interesting remaining topics is the definition of "RISC." Don't be fooled into thinking there is one definition of RISC. The best I have heard so far is from John Mashey: "RISC is a label most commonly used for a set of instruction set architecture characteristics chosen to ease the use of aggressive implementation techniques found in high performance processors (regardless of RISC, CISC, or irrelevant)."

This chapter is about CISC and RISC instruction set architectures and the differences between them. We also describe newer processors that can

execute more than one instruction at a time and can execute instructions out of order.

Why CISC?

You might ask, “If RISC is faster, why did people bother with CISC designs in the first place?” The short answer is that in the beginning, CISC *was* the right way to go; RISC wasn’t always both feasible and affordable. Every kind of design incorporates trade-offs, and over time, the best systems will make them differently. In the past, the design variables favored CISC.

Space and Time

To start, we’ll ask you how well you know the assembly language for your work- station. The answer is probably that you haven’t even seen it. Why bother? Compilers and development tools are very good, and if you have a problem, you can debug it at the source level. However, 30 years ago, “respectable” programmers understood the machine’s instruction set. High-level language compilers were commonly available, but they didn’t generate the fastest code, and they weren’t terribly thrifty with memory. When programming, you needed to save both space and time, which meant you knew how to program in assembly language. Accordingly, you could develop an opinion about the machine’s instruction set. A good instruction set was both easy to use and powerful. In many ways these qualities were the same: “powerful” instructions accomplished a lot, and saved the programmer from specifying many little steps — which, in turn, made them easy to use. But they had other, less apparent (though perhaps more important) features as well: powerful instructions saved memory and time.

Back then, computers had very little storage by today’s standards. An instruction that could roll all the steps of a complex operation, such as a do-loop, into single opcode[footnote] was a plus, because memory was precious. To put some stakes in the ground, consider the last vacuum-tube computer that IBM built, the model 704 (1956). It had hardware floating-point, including a division operation, index registers, and instructions that could operate directly on memory locations. For instance, you could add two numbers together and store the result back into memory with a single command. The Philco 2000, an early transistorized machine (1959), had an operation that could repeat a sequence of instructions until the contents of a counter was decremented to zero — very much like a do-loop. These were

complex operations, even by today's standards. However, both machines had a limited amount of memory — 32-K words. The less memory your program took up, the more you had available for data, and the less likely that you would have to resort to overlaying portions of the program on top of one another.
Opcode = operation code = instruction.

Complex instructions saved time, too. Almost every large computer following the IBM 704 had a memory system that was slower than its central processing unit (CPU). When a single instruction can perform several operations, the overall number of instructions retrieved from memory can be reduced. Minimizing the number of instructions was particularly important because, with few exceptions, the machines of the late 1950s were very sequential; not until the current instruction was completed did the computer initiate the process of going out to memory to get the next instruction.[footnote] By contrast, modern machines form something of a bucket brigade — passing instructions in from memory and figuring out what they do on the way — so there are fewer gaps in processing.
In 1955, IBM began constructing a machine known as Stretch. It was the first computer to process several instructions at a time in stages, so that they streamed in, rather than being fetched in a piece- meal fashion. The goal was to make it 25 times faster than the then brand-new IBM 704. It was six years before the first Stretch was delivered to Los Alamos National Laboratory. It was indeed faster, but it was expensive to build. Eight were sold for a loss of $20 million.

If the designers of early machines had had very fast and abundant instruction memory, sophisticated compilers, and the wherewithal to build the instruction "bucket brigade" — cheaply — they might have chosen to create machines with simple instruction sets. At the time, however, technological choices indicated that instructions should be powerful and thrifty with memory.

Beliefs About Complex Instruction Sets

So, given that the lot was cast in favor of complex instruction sets, computer architects had license to experiment with matching them to the intended purposes of the machines. For instance, the do-loop instruction on the Philco 2000 looked like a good companion for procedural languages like FORTRAN. Machine designers assumed that compiler writers could generate object programs using these powerful machine instructions, or possibly that the compiler could be eliminated, and that the machine could execute source code directly in hardware.

You can imagine how these ideas set the tone for product marketing. Up until the early 1980s, it was common practice to equate a bigger instruction set with a more powerful computer. When clock speeds were increasing by multiples, no increase in instruction set complexity could fetter a new model of computer enough so that there wasn’t still a tremendous net increase in speed. CISC machines kept getting faster, in spite of the increased operation complexity.

As it turned out, assembly language programmers used the complicated machine instructions, but compilers generally did not. It was difficult enough to get a compiler to recognize when a complicated instruction could be used, but the real problem was one of optimizations: verbatim translation of source constructs isn’t very efficient. An optimizing compiler works by simplifying and eliminating redundant computations. After a pass through an optimizing compiler, opportunities to use the complicated instructions tend to disappear.

Fundamental of RISC

A RISC machine could have been built in 1960. (In fact, Seymour Cray built one in 1964 — the CDC 6600.) However, given the same costs of components, technical barriers, and even expectations for how computers would be used, you would probably still have chosen a CISC design — even with the benefit of hindsight.

The exact inspiration that led to developing high performance RISC microprocessors in the 1980s is a subject of some debate. Regardless of the motivation of the RISC designers, there were several obvious pressures that affected the development of RISC:

- The number of transistors that could fit on a single chip was increasing. It was clear that one would eventually be able to fit all the components from a processor board onto a single chip.
- Techniques such as pipelining were being explored to improve performance. Variable-length instructions and variable-length instruction execution times (due to varying numbers of microcode steps) made implementing pipelines more difficult.
- As compilers improved, they found that well-optimized sequences of stream- lined instructions often outperformed the equivalent complicated multi-cycle instructions. (See Appendix A, Processor Architectures, and Appendix B, Looking at Assembly Language.)

The RISC designers sought to create a high performance single-chip processor with a fast clock rate. When a CPU can fit on a single chip, its cost is decreased, its reliability is increased, and its clock speed can be increased. While not all RISC processors are single-chip implementation, most use a single chip.

To accomplish this task, it was necessary to discard the existing CISC instruction sets and develop a new minimal instruction set that could fit on a single chip. Hence the term *reduced instruction set computer*. In a sense reducing the instruction set was not an “end” but a means to an end.

For the first generation of RISC chips, the restrictions on the number of components that could be manufactured on a single chip were severe,

forcing the designers to leave out hardware support for some instructions. The earliest RISC processors had no floating-point support in hardware, and some did not even support integer multiply in hardware. However, these instructions could be implemented using software routines that combined other instructions (a microcode of sorts).

These earliest RISC processors (most severely reduced) were not overwhelming successes for four reasons:

- It took time for compilers, operating systems, and user software to be retuned to take advantage of the new processors.
- If an application depended on the performance of one of the software-implemented instructions, its performance suffered dramatically.
- Because RISC instructions were simpler, more instructions were needed to accomplish the task.
- Because all the RISC instructions were 32 bits long, and commonly used CISC instructions were as short as 8 bits, RISC program executables were often larger.

As a result of these last two issues, a RISC program may have to fetch more memory for its instructions than a CISC program. This increased appetite for instructions actually clogged the memory bottleneck until sufficient caches were added to the RISC processors. In some sense, you could view the caches on RISC processors as the microcode store in a CISC processor. Both reduced the overall appetite for instructions that were loaded from memory.

While the RISC processor designers worked out these issues and the manufacturing capability improved, there was a battle between the existing (now called CISC) processors and the new RISC (not yet successful) processors. The CISC processor designers had mature designs and well-tuned popular software. They also kept adding performance tricks to their systems. By the time Motorola had evolved from the MC68000 in 1982 that was a CISC processor to the MC68040 in 1989, they referred to the MC68040 as a RISC processor.[footnote]
And they did it without ever taking out a single instruction!

However, the RISC processors eventually became successful. As the amount of logic available on a single chip increased, floating-point operations were added back onto the chip. Some of the additional logic was used to add on-chip cache to solve some of the memory bottleneck problems due to the larger appetite for instruction memory. These and other changes moved the RISC architectures from the defensive to the offensive.

RISC processors quickly became known for their affordable high-speed floating- point capability compared to CISC processors.[footnote] This excellent performance on scientific and engineering applications effectively created a new type of computer system, the workstation. Workstations were more expensive than personal computers but their cost was sufficiently low that workstations were heavily used in the CAD, graphics, and design areas. The emerging workstation market effectively created three new computer companies in Apollo, Sun Microsystems, and Silicon Graphics.
The typical CISC microprocessor in the 1980s supported floating-point operations in a separate coprocessor.

Some of the existing companies have created competitive RISC processors in addition to their CISC designs. IBM developed its RS-6000 (RIOS) processor, which had excellent floating-point performance. The Alpha from DEC has excellent performance in a number of computing benchmarks. Hewlett-Packard has developed the PA-RISC series of processors with excellent performance. Motorola and IBM have teamed to develop the PowerPC series of RISC processors that are used in IBM and Apple systems.

By the end of the RISC revolution, the performance of RISC processors was so impressive that single and multiprocessor RISC-based server systems quickly took over the minicomputer market and are currently encroaching on the traditional mainframe market.

Characterizing RISC

RISC is more of a design philosophy than a set of goals. Of course every RISC processor has its own personality. However, there are a number of features commonly found in machines people consider to be RISC:

- Instruction pipelining
- Pipelining floating-point execution
- Uniform instruction length
- Delayed branching
- Load/store architecture
- Simple addressing modes

This list highlights the differences between RISC and CISC processors. Naturally, the two types of instruction-set architectures have much in common; each uses registers, memory, etc. And many of these techniques are used in CISC machines too, such as caches and instruction pipelines. It is the fundamental differences that give RISC its speed advantage: focusing on a smaller set of less powerful instructions makes it possible to build a faster computer.

However, the notion that RISC machines are generally simpler than CISC machines isn't correct. Other features, such as functional pipelines, sophisticated memory systems, and the ability to issue two or more instructions per clock make the latest RISC processors the most complicated ever built. Furthermore, much of the complexity that has been lifted from the instruction set has been driven into the compilers, making a good optimizing compiler a prerequisite for machine performance.

Let's put ourselves in the role of computer architect again and look at each item in the list above to understand why it's important.

Pipelines

Everything within a digital computer (RISC or CISC) happens in step with a *clock*: a signal that paces the computer's circuitry. The rate of the clock, or *clock speed*, determines the overall speed of the processor. There is an upper limit to how fast you can clock a given computer.

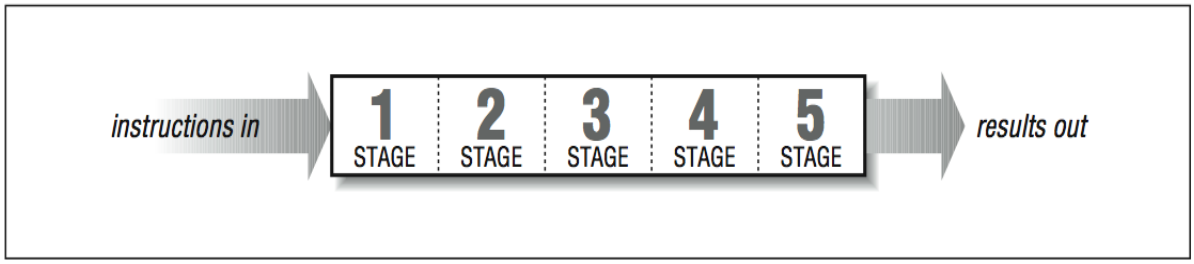
A number of parameters place an upper limit on the clock speed, including the semiconductor technology, packaging, the length of wires tying the pieces together, and the longest path in the processor. Although it may be possible to reach blazing speed by optimizing all of the parameters, the cost

can be prohibitive. Furthermore, exotic computers don't make good office mates; they can require too much power, produce too much noise and heat, or be too large. There is incentive for manufacturers to stick with manufacturable and marketable technologies.

Reducing the number of clock ticks it takes to execute an individual instruction is a good idea, though cost and practicality become issues beyond a certain point. A greater benefit comes from partially overlapping instructions so that more than one can be in progress simultaneously. For instance, if you have two additions to perform, it would be nice to execute them both at the same time. How do you do that? The first, and perhaps most obvious, approach, would be to start them simultaneously. Two additions would execute together and complete together in the amount of time it takes to perform one. As a result, the throughput would be effectively doubled. The downside is that you would need hardware for two adders in a situation where space is usually at a premium (especially for the early RISC processors).

Other approaches for overlapping execution are more cost-effective than side-by-side execution. Imagine what it would be like if, a moment after launching one operation, you could launch another without waiting for the first to complete. Perhaps you could start another of the same type right behind the first one — like the two additions. This would give you nearly the performance of side-by-side execution without duplicated hardware. Such a mechanism does exist to varying degrees in all computers — CISC and RISC. It's called a pipeline. A pipeline takes advantage of the fact that many operations are divided into identifiable steps, each of which uses different resources on the processor.[footnote]
Here is a simple analogy: imagine a line at a fast-food drive up window. If there is only one window, one customer orders and pays, and the food is bagged and delivered to the customer before the second customer orders. For busier restaurants, there are three windows. First you order, then move ahead. Then at a second window, you pay and move ahead. At the third window you pull up, grab the food and roar off into the distance. While your wait at the three-window (pipelined) drive-up may have been slightly longer than your wait at the one-window (non-pipelined) restaurant, the

pipeline solution is significantly better because multiple customers are being processed simultaneously.
A Pipeline



[link] shows a conceptual diagram of a pipeline. An operation entering at the left proceeds on its own for five clock ticks before emerging at the right. Given that the pipeline stages are independent of one another, up to five operations can be in flight at a time as long as each instruction is delayed long enough for the previous instruction to clear the pipeline stage. Consider how powerful this mechanism is: where before it would have taken five clock ticks to get a single result, a pipeline produces as much as one result every clock tick.

Pipelining is useful when a procedure can be divided into stages. Instruction processing fits into that category. The job of retrieving an instruction from memory, figuring out what it does, and doing it are separate steps we usually lump together when we talk about executing an instruction. The number of steps varies, depending on whose processor you are using, but for illustration, let's say there are five:

1. *Instruction fetch*: The processor fetches an instruction from memory.
2. *Instruction decode*: The instruction is recognized or decoded.
3. *Operand Fetch*: The processor fetches the operands the instruction needs. These operands may be in registers or in memory.
4. *Execute*: The instruction gets executed.
5. *Writeback*: The processor writes the results back to wherever they are supposed to go —possibly registers, possibly memory.

Ideally, instruction

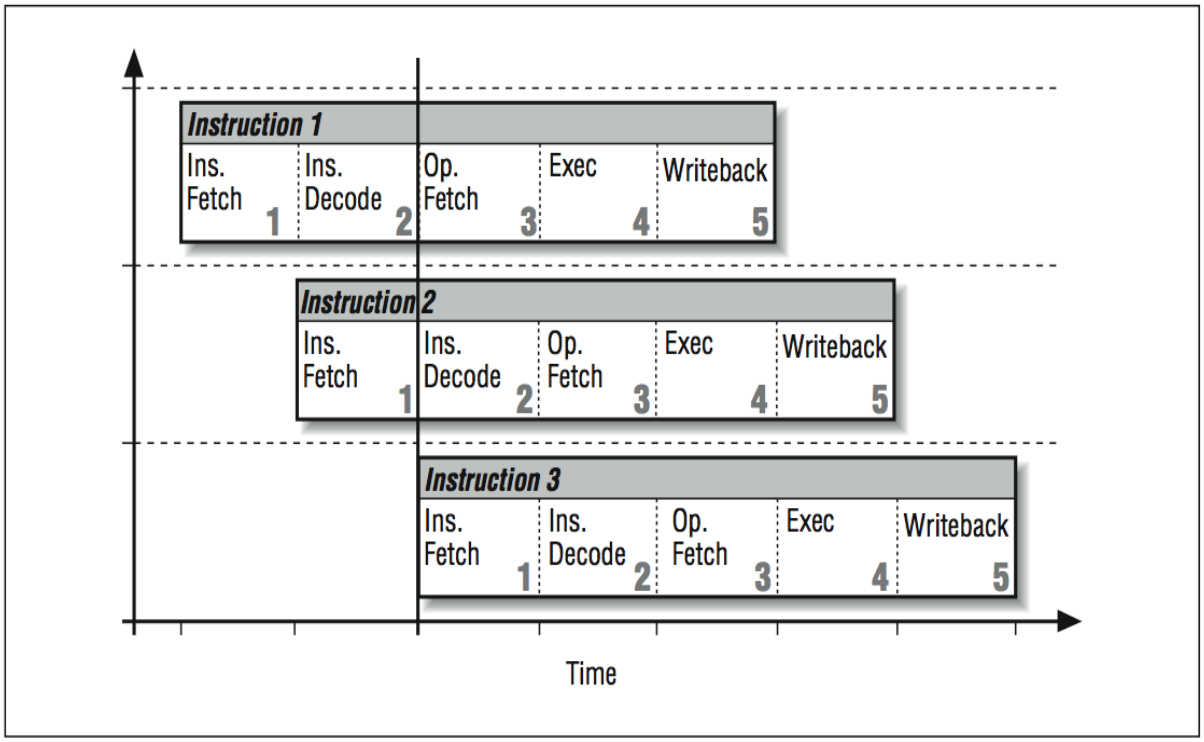
1. Will be entering the operand fetch stage as instruction

2. enters instruction decode stage and instruction
3. starts instruction fetch, and so on.

Our pipeline is five stages deep, so it should be possible to get five instructions in flight all at once. If we could keep it up, we would see one instruction complete per clock cycle.

Simple as this illustration seems, instruction pipelining is complicated in real life. Each step must be able to occur on different instructions simultaneously, and delays in any stage have to be coordinated with all those that follow. In [link] we see three instructions being executed simultaneously by the processor, with each instruction in a different stage of execution.

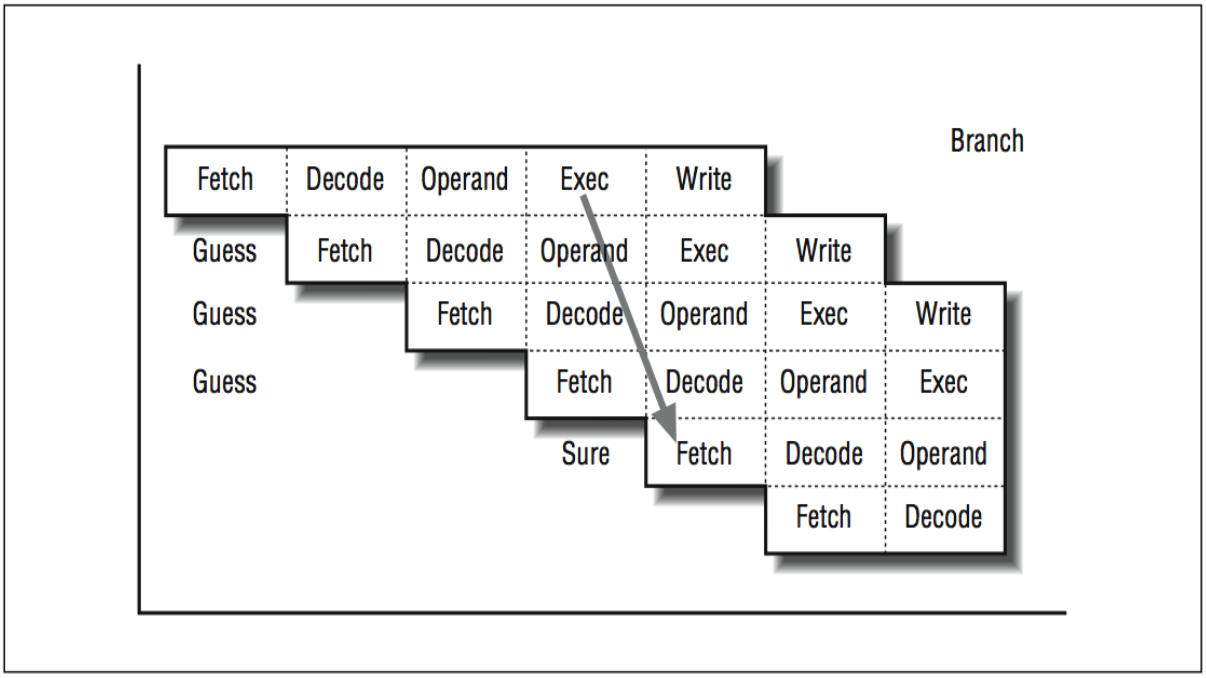
Three instructions in flight through one pipeline



For instance, if a complicated memory access occurs in stage three, the instruction needs to be delayed before going on to stage four because it takes some time to calculate the operand's address and retrieve it from memory. All the while, the rest of the pipeline is stalled. A simpler instruction, sitting in one of the earlier stages, can't continue until the traffic ahead clears up.

Now imagine how a jump to a new program address, perhaps caused by an if statement, could disrupt the pipeline flow. The processor doesn't know an instruction is a branch until the decode stage. It usually doesn't know whether a branch will be taken or not until the execute stage. As shown in [link], during the four cycles after the branch instruction was fetched, the processor blindly fetches instructions sequentially and starts these instructions through the pipeline.

Detecting a branch



If the branch "falls through," then everything is in great shape; the pipeline simply executes the next instruction. It's as if the branch were a "no-op" instruction. However, if the branch jumps away, those three partially processed instructions never get executed. The first order of business is to discard these "in-flight" instructions from the pipeline. It turns out that because none of these instructions was actually going to do anything until its execute stage, we can throw them away without hurting anything (other than our efficiency). Somehow the processor has to be able to clear out the pipeline and restart the pipeline at the branch destination.

Unfortunately, branch instructions occur every five to ten instructions in many programs. If we executed a branch every fifth instruction and only

half our branches fell through, the lost efficiency due to restarting the pipeline after the branches would be 20 percent.

You need optimal conditions to keep the pipeline moving. Even in less-than-optimal conditions, instruction pipelining is a big win — especially for RISC processors. Interestingly, the idea dates back to the late 1950s and early 1960s with the UNI- VAC LARC and the IBM Stretch. Instruction pipelining became mainstreamed in 1964, when the CDC 6600 and the IBM S/360 families were introduced with pipelined instruction units — on machines that represented RISC-ish and CISC designs, respectively. To this day, ever more sophisticated techniques are being applied to instruction pipelining, as machines that can overlap instruction execution become commonplace.

Pipelined Floating-Point Operations

Because the execution stage for floating-point operations can take longer than the execution stage for fixed-point computations, these operations are typically pipelined, too. Generally, this includes floating-point addition, subtraction, multiplication, comparisons, and conversions, though it might not include square roots and division. Once a pipelined floating-point operation is started, calculations continue through the several stages without delaying the rest of the processor. The result appears in a register at some point in the future.

Some processors are limited in the amount of overlap their floating-point pipelines can support. Internal components of the pipelines may be shared (for adding, multiplying, normalizing, and rounding intermediate results), forcing restrictions on when and how often you can begin new operations. In other cases, floating- point operations can be started every cycle regardless of the previous floating- point operations. We say that such operations are *fully pipelined*.

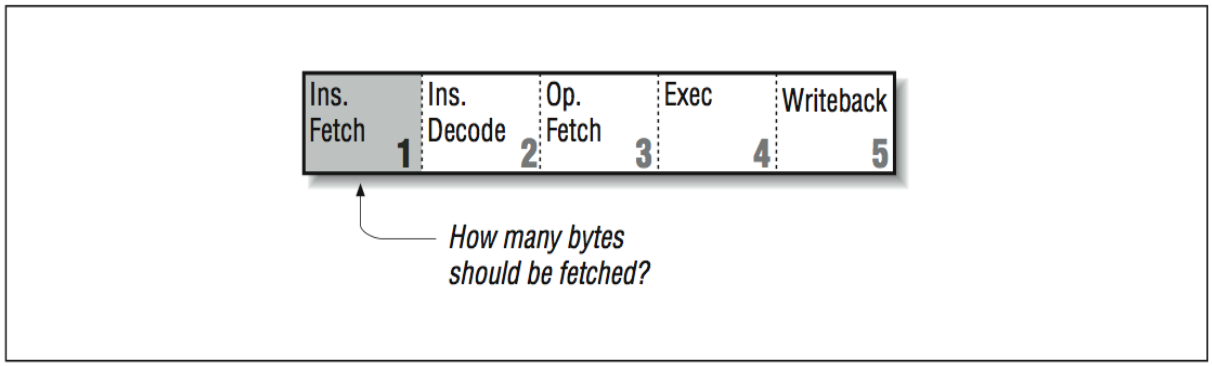
The number of stages in floating-point pipelines for affordable computers has decreased over the last 10 years. More transistors and newer algorithms make it possible to perform a floating-point addition or multiplication in just one to three cycles. Generally the most difficult instruction to perform

in a single cycle is the floating-point multiply. However, if you dedicate enough hardware to it, there are designs that can operate in a single cycle at a moderate clock rate.

Uniform Instruction Length

Our sample instruction pipeline had five stages: instruction fetch, instruction decode, operand fetch, execution, and writeback. We want this pipeline to be able to process five instructions in various stages without stalling. Decomposing each operation into five identifiable parts, each of which is roughly the same amount of time, is challenging enough for a RISC computer. For a designer working with a CISC instruction set, it's especially difficult because CISC instructions come in varying lengths. A simple "return from subroutine" instruction might be one byte long, for instance, whereas it would take a longer instruction to say "add register four to memory location 2005 and leave the result in register five." The number of bytes to be fetched must be known by the fetch stage of the pipeline as shown in [link].

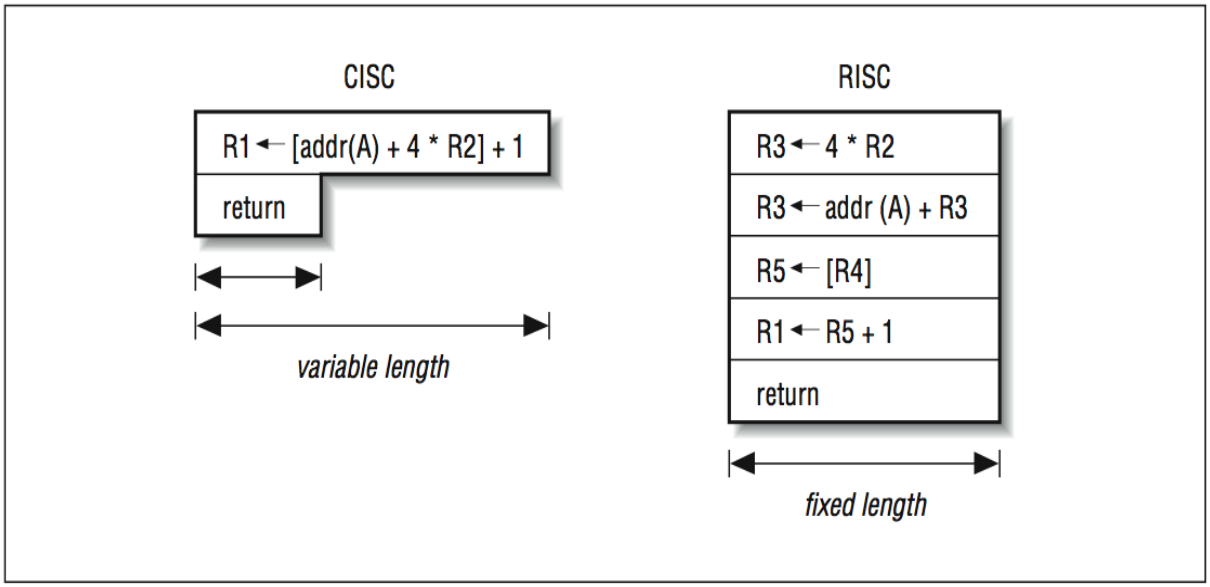
Variable length instructions make pipelining difficult



The processor has no way of knowing how long an instruction will be until it reaches the decode stage and determines what it is. If it turns out to be a long instruction, the processor may have to go back to memory and get the portion left behind; this stalls the pipeline. We could eliminate the problem by requiring that all instructions be the same length, and that there be a limited number of instruction formats as shown in [link]. This way, every instruction entering the pipeline is known *a priori* to be complete — not needing another memory access. It would also be easier for the processor to

locate the instruction fields that specify registers or constants. Altogether because RISC can assume a fixed instruction length, the pipeline flows much more smoothly.

Variable-length CISC versus fixed-length RISC instructions



Delayed Branches

As described earlier, branches are a significant problem in a pipelined architecture. Rather than take a penalty for cleaning out the pipeline after a misguessed branch, many RISC designs require an instruction after the branch. This instruction, in what is called the *branch delay slot*, is executed no matter what way the branch goes. An instruction in this position should be useful, or at least harmless, whichever way the branch proceeds. That is, you expect the processor to execute the instruction following the branch in either case, and plan for it. In a pinch, a no-op can be used. A slight variation would be to give the processor the ability to *annul* (or squash) the instruction appearing in the branch delay slot if it turns out that it shouldn't have been issued after all:

```
      ADD       R1,R2,R1            add r1 to r2 and
store in r1
      SUB       R3,R1,R3            subtract r1 from
```

```
r3, store in r3
      BRA      SOMEWHERE          branch somewhere
else
      LABEL1   ZERO R3            instruction in
branch delay slot
                ...
```

While branch delay slots appeared to be a very clever solution to eliminating pipeline stalls associated with branch operations, as processors moved toward exe- cuting two and four instructions simultaneously, another approach was needed.[footnote]
Interestingly, while the delay slot is no longer critical in processors that execute four instructions simultaneously, there is not yet a strong reason to remove the feature. Removing the delay slot would be nonupwards-compatible, breaking many existing codes. To some degree, the branch delay slot has become “baggage” on those “new” 10-year-old architectures that must continue to support it.

A more robust way of eliminating pipeline stalls was to “predict” the direction of the branch using a table stored in the decode unit. As part of the decode stage, the CPU would notice that the instruction was a branch and consult a table that kept the recent behavior of the branch; it would then make a guess. Based on the guess, the CPU would immediately begin fetching at the predicted location. As long as the guesses were correct, branches cost exactly the same as any other instruction.

If the prediction was wrong, the instructions that were in process had to be can- celled, resulting in wasted time and effort. A simple branch prediction scheme is typically correct well over 90% of the time, significantly reducing the overall negative performance impact of pipeline stalls due to branches. All recent RISC designs incorporate some type of branch prediction, making branch delay slots effectively unnecessary.

Another mechanism for reducing branch penalties is *conditional execution*. These are instructions that look like branches in source code, but turn out to be a special type of instruction in the object code. They are very useful because they replace test and branch sequences altogether. The following lines of code capture the sense of a conditional branch:

```
IF ( B < C ) THEN
   A = D
 ELSE
   A = E
 ENDIF
```

Using branches, this would require at least two branches to ensure that the proper value ended up in A. Using conditional execution, one might generate code that looks as follows:

```
COMPARE  B < C
IF TRUE  A = D     conditional instruction
IF FALSE A = E     conditional instruction
```

This is a sequence of three instructions with *no branches*. One of the two assignments executes, and the other acts as a no-op. No branch prediction is needed, and the pipeline operates perfectly. There is a cost to taking this approach when there are a large number of instructions in one or the other branch paths that would seldom get executed using the traditional branch instruction model.

Load/Store Architecture

In a load/store instruction set architecture, memory references are limited to explicit load and store instructions. Each instruction may not make more than one memory reference per instruction. In a CISC processor, arithmetic and logical instructions can include embedded memory references. There are three reasons why limiting loads and stores to their own instructions is an improvement:

- First, we want all instructions to be the same length, for the reasons given above. However, fixed lengths impose a budget limit when it comes to describing what the operation does and which registers it uses. An instruction that both referenced memory and performed some calculation wouldn't fit within one instruction word.
- Second, giving every instruction the option to reference memory would com- plicate the pipeline because there would be two computations to perform— the address calculation plus whatever the instruction is supposed to do — but there is only one execution stage. We could throw more hardware at it, but by restricting memory references to explicit loads and stores, we can avoid the problem entirely. Any instruction can perform an address calculation or some other operation, but no instruction can do both.
- The third reason for limiting memory references to explicit loads and stores is that they can take more time than other instructions — sometimes two or three clock cycles more. A general instruction with an embedded memory reference would get hung up in the operand fetch stage for those extra cycles, waiting for the reference to complete. Again we would be faced with an instruction pipeline stall.

Explicit load and store instructions can kick off memory references in the pipeline's execute stage, to be completed at a later time (they might complete immediately; it depends on the processor and the cache). An operation downstream may require the result of the reference, but that's all right, as long as it is far enough downstream that the reference has had time to complete.

Simple Addressing Modes

Just as we want to simplify the instruction set, we also want a simple set of memory addressing modes. The reasons are the same: complicated address calculations, or those that require multiple memory references, will take too much time and stall the pipeline. This doesn't mean that your program can't use elegant data structures; the compiler explicitly generates the extra address arithmetic when it needs it, as long as it can count on a few fundamental addressing modes in hardware. In fact, the extra address

arithmetic is often easier for the compiler to optimize into faster forms (see [link] and [link]).

Of course, cutting back the number of addressing modes means that some memory references will take more real instructions than they might have taken on a CISC machine. However, because everything executes more quickly, it generally is still a performance win.

Second-Generation RISC Processors

The Holy Grail for early RISC machines was to achieve one instruction per clock. The idealized RISC computer running at, say, 50 MHz, would be able to issue 50 million instructions per second assuming perfect pipeline scheduling. As we have seen, a single instruction will take five or more clock ticks to get through the instruction pipeline, but if the pipeline can be kept full, the aggregate rate will, in fact, approach one instruction per clock. Once the basic pipelined RISC processor designs became successful, competition ensued to determine which company could build the best RISC processor.

Second-generation RISC designers used three basic methods to develop competitive RISC processors:

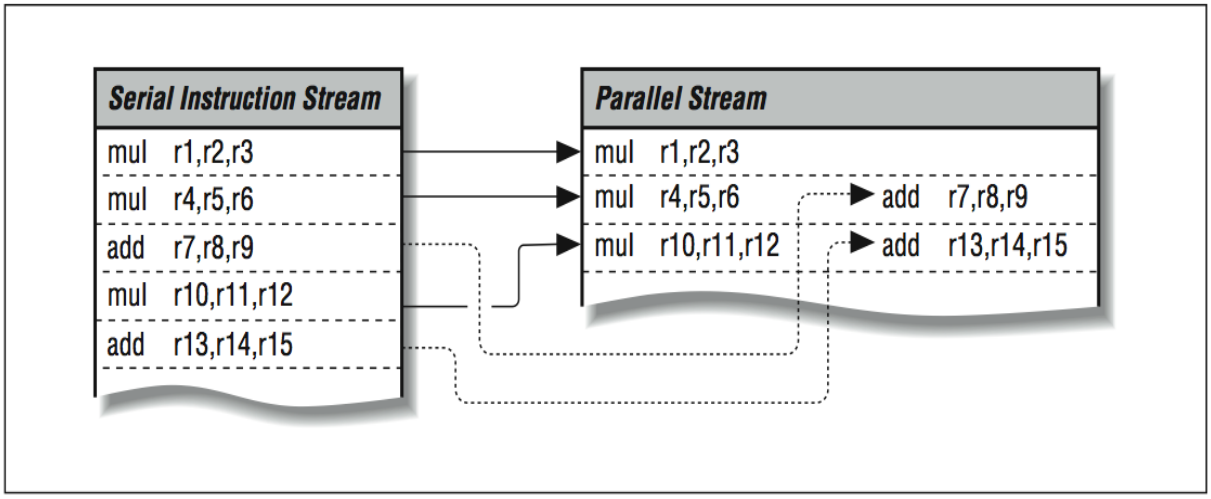
- Improve the manufacturing processes to simply make the clock rate faster. Take a simple design; make it smaller and faster. This approach was taken by the Alpha processors from DEC. Alpha processors typically have had clock rates double those of the closest competitor.
- Add duplicate compute elements on the space available as we can manufacture chips with more transistors. This could allow two instructions to be executed per cycle and could double performance without increasing clock rate. This technique is called superscalar.
- Increase the number of stages in the pipeline above five. If the instructions can truly be decomposed evenly into, say, ten stages, the clock rate could theoretically be doubled without requiring new manufacturing processes. This technique was called *superpipelining*. The MIPS processors used this technique with some success.

Superscalar Processors

The way you get two or more instructions per clock is by starting several operations side by side, possibly in separate pipelines. In [link], if you have an integer addition and a multiplication to perform, it should be possible to begin them simultaneously, provided they are independent of each other (as long as the multiplication does not need the output of the addition as one of its operands or vice versa). You could also execute multiple fixed-point

instructions — compares, integer additions, etc. — at the same time, provided that they, too, are independent. Another term used to describe superscalar processors is *multiple instruction issue* processors.

Decomposing a serial stream



The number and variety of operations that can be run in parallel depends on both the program and the processor. The program has to have enough usable parallelism so that there are multiple things to do, and the processor has to have an appropriate assortment of functional units and the ability to keep them busy. The idea is conceptually simple, but it can be a challenge for both hardware designers and compiler writers. Every opportunity to do several things in parallel exposes the danger of violating some precedence (i.e., performing computations in the wrong order).

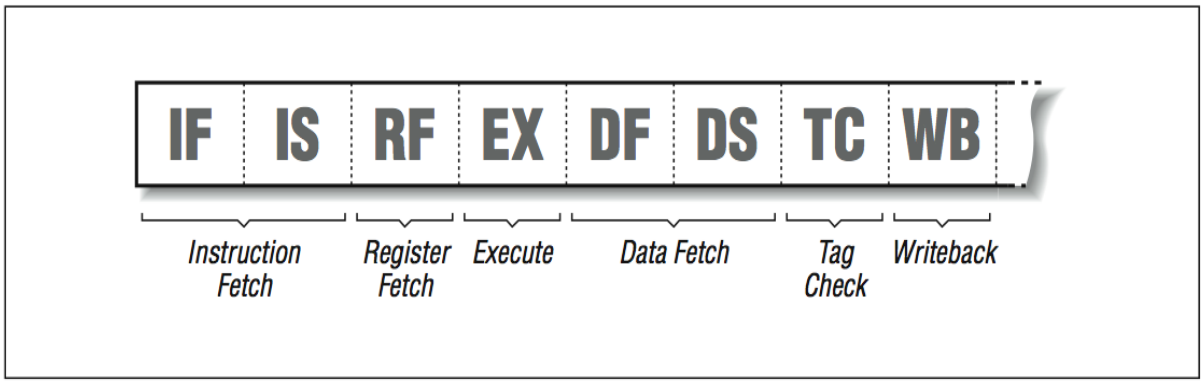
Superpipelined Processors

Roughly stated, simpler circuitry can run at higher clock speeds. Put yourself in the role of a CPU designer again. Looking at the instruction pipeline of your processor, you might decide that the reason you can't get more speed out of it is that some of the stages are too complicated or have too much going on, and they are placing limits on how fast the whole pipeline can go. Because the stages are clocked in unison, the slowest of them forms a weak link in the chain.

If you divide the complicated stages into less complicated portions, you can increase the overall speed of the pipeline. This is called *superpipelining*.

More instruction pipeline stages with less complexity per stage will do the same work as a pipelined processor, but with higher throughput due to increased clock speed. [link] shows an eight-stage pipeline used in the MIPS R4000 processor.

MIPS R4000 instruction pipeline



Theoretically, if the reduced complexity allows the processor to clock faster, you can achieve nearly the same performance as superscalar processors, yet without instruction mix preferences. For illustration, picture a superscalar processor with two units — fixed- and floating-point — executing a program that is composed solely of fixed-point calculations; the floating-point unit goes unused. This reduces the superscalar performance by one half compared to its theoretical maximum. A superpipelined processor, on the other hand, will be perfectly happy to handle an unbalanced instruction mix at full speed.

Superpipelines are not new; deep pipelines have been employed in the past, notably on the CDC 6600. The label is a marketing creation to draw contrast to superscalar processing, and other forms of efficient, high-speed computing.

Superpipelining can be combined with other approaches. You could have a superscalar machine with deep pipelines (DEC AXP and MIPS R-8000 are examples). In fact, you should probably expect that faster pipelines with more stages will become so commonplace that nobody will remember to call them superpipelines after a while.

RISC Means Fast

We all know that the “R” in RISC means “reduced.” Lately, as the number of components that can be manufactured on a chip has increased, CPU designers have been looking at ways to make their processors faster by adding features. We have already talked about many of the features such as on-chip multipliers, very fast floating-point, lots of registers, and on-chip caches. Even with all of these features, there seems to be space left over. Also, because much of the design of the control section of the processor is automated, it might not be so bad to add just a “few” new instructions here and there. Especially if simulations indicate a 10% overall increase in speed!

So, what does it mean when they add 15 instructions to a RISC instruction set architecture (ISA)? Would we call it “not-so-RISC”? A suggested term for this trend is FISC, or *fast instruction set computer*. The point is that reducing the number of instructions is not the goal. The goal is to build the fastest possible processor within the manufacturing and cost constraints. [footnote]
People will argue forever but, in a sense, reducing the instruction set was never an end in itself, it was a means to an end.

Some of the types of instructions that are being added into architectures include:

- More addressing modes
- Meta-instructions such as “decrement counter and branch if non-zero”
- Specialized graphics instructions such as the Sun VIS set, the HP graphics instructions, the MIPS Digital Media Extentions (MDMX), and the Intel MMX instructions

Interestingly, the reason that the first two are feasible is that adder units take up so little space, it is possible to put one adder into the decode unit and another into the load/store unit. Most visualization instruction sets take up very little chip area. They often provide “ganged” 8-bit computations to allow a 64-bit register to be used to perform eight 8-bit operations in a single instruction.

Out-of-Order Execution: The Post-RISC Architecture

We're never satisfied with the performance level of our computing equipment and neither are the processor designers. Two-way superscalar processors were very successful around 1994. Many designs were able to execute 1.6–1.8 instructions per cycle on average, using all of the tricks described so far. As we became able to manufacture chips with an ever-increasing transistor count, it seemed that we would naturally progress to four-way and then eight-way superscalar processors. The fundamental problem we face when trying to keep four functional units busy is that it's difficult to find contiguous sets of four (or eight) instructions that can be executed in parallel. It's an easy cop-out to say, "the compiler will solve it all."

The solution to these problems that will allow these processors to effectively use four functional units per cycle and hide memory latency is *out-of-order execution* and *speculative execution*. Out-of-order execution allows a later instruction to be processed before an earlier instruction is completed. The processor is "betting" that the instruction will execute, and the processor will have the precomputed "answer" the instruction needs. In some ways, portions of the RISC design philosophy are turned inside-out in these new processors.

Speculative Computation

To understand the post-RISC architecture, it is important to separate the concept of computing a value for an instruction and actually executing the instruction. Let's look at a simple example:

```
LD    R10,R2(R0)   Load into R10 from memory
 ...               30 Instructions of various
kinds (not FDIV)
FDIV  R4,R5,R6     R4 = R5 / R6
```

Assume that (1) we are executing the load instruction, (2) R5 and R6 are already loaded from earlier instructions, (3) it takes 30 cycles to do a floating-point divide, and (4) there are no instructions that need the divide unit between the LD and the FDIV. Why not start the divide unit *computing* the FDIV right now, storing the result in some temporary scratch area? It has nothing better to do. When or if we arrive at the FDIV, we will know the result of the calculation, copy the scratch area into R4, and the FDIV will appear to *execute* in one cycle. Sound farfetched? Not for a post-RISC processor.

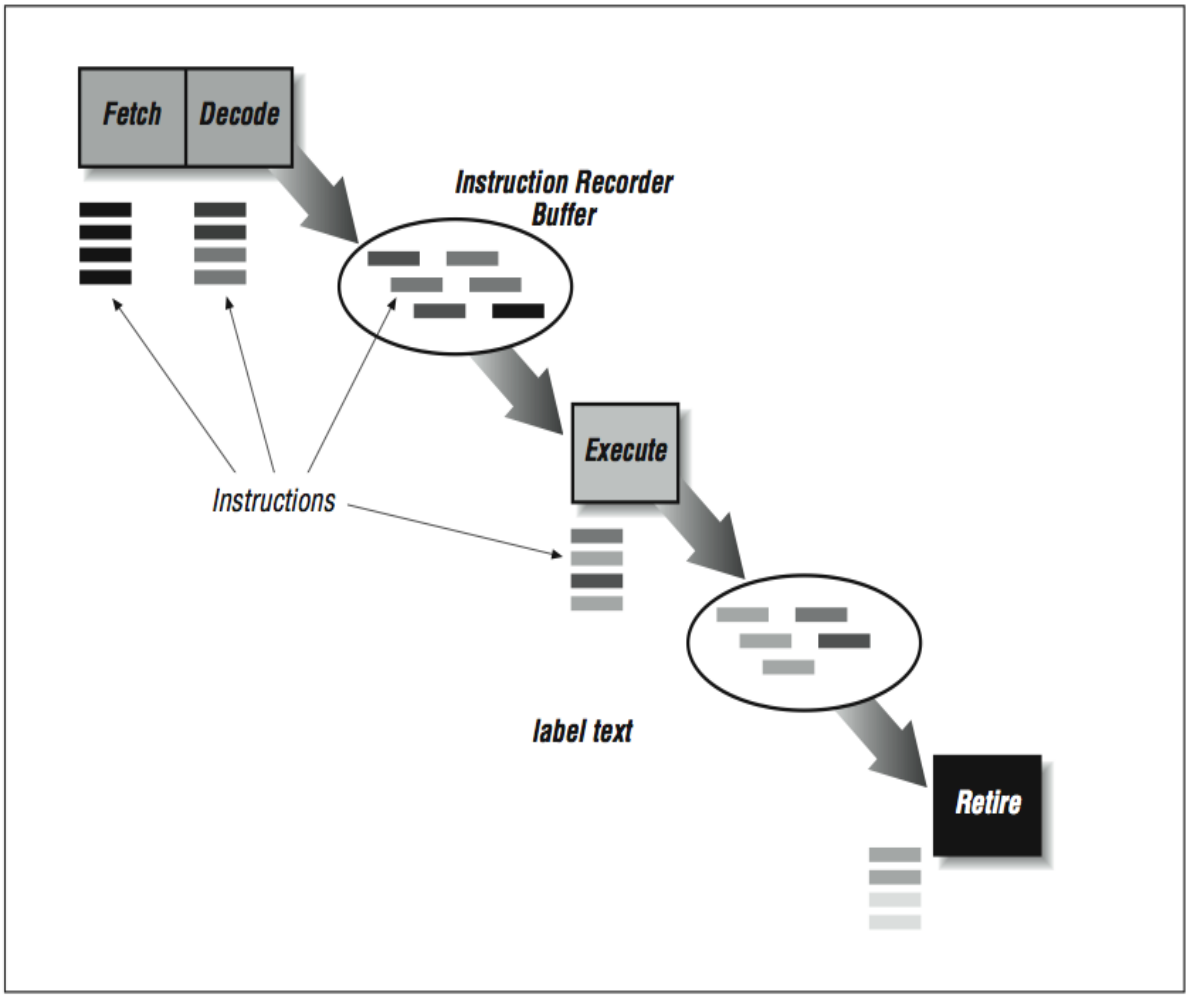
The post-RISC processor must be able to speculatively compute results before the processor knows whether or not an instruction will actually execute. It accomplishes this by allowing instructions to start that will never finish and allowing later instructions to start before earlier instructions finish.

To store these instructions that are in limbo between started and finished, the post-RISC processor needs some space on the processor. This space for instructions is called the *instruction reorder buffer* (IRB).

The Post-RISC Pipeline

The post-RISC processor pipeline in [link] looks somewhat different from the RISC pipeline. The first two stages are still instruction fetch and decode. Decode includes branch prediction using a table that indicates the probable behavior of a branch. Once instructions are decoded and branches are predicted, the instructions are placed into the IRB to be computed as soon as possible.

Post-RISC pipeline



The IRB holds up to 60 or so instructions that are waiting to execute for one reason or another. In a sense, the fetch and decode/predict phases operate until the buffer fills up. Each time the decode unit predicts a branch, the following instructions are marked with a different indicator so they can be found easily if the prediction turns out to be wrong. Within the buffer, instructions are allowed to go to the computational units when the instruction has all of its operand values. Because the instructions are computing results without being executed, any instruction that has its input values and an available computation unit can be computed. The results of these computations are stored in extra registers not visible to the programmer called *rename registers*. The processor allocates rename registers, as they are needed for instructions being computed.

The execution units may have one or more pipeline stages, depending on the type of the instruction. This part looks very much like traditional superscalar RISC processors. Typically up to four instructions can begin computation from the IRB in any cycle, provided four instructions are available with input operands and there are sufficient computational units for those instructions.

Once the results for the instruction have been computed and stored in a rename register, the instruction must wait until the preceding instructions finish so we know that the instruction actually executes. In addition to the computed results, each instruction has flags associated with it, such as exceptions. For example, you would not be happy if your program crashed with the following message: “Error, divide by zero. I was precomputing a divide in case you got to the instruction to save some time, but the branch was mispredicted and it turned out that you were never going to execute that divide anyway. I still had to blow you up though. No hard feelings? Signed, the post-RISC CPU.” So when a speculatively computed instruction divides by zero, the CPU must simply store that fact until it knows the instruction will execute and at that moment, the program can be legitimately crashed.

If a branch does get mispredicted, a lot of bookkeeping must occur very quickly. A message is sent to all the units to discard instructions that are part of all control flow paths beyond the incorrect branch.

Instead of calling the last phase of the pipeline “writeback,” it’s called “retire.” The retire phase is what “executes” the instructions that have already been computed. The retire phase keeps track of the instruction execution order and retires the instructions in program order, posting results from the rename registers to the actual registers and raising exceptions as necessary. Typically up to four instructions can be retired per cycle.

So the post-RISC pipeline is actually three pipelines connected by two buffers that allow instructions to be processed out of order. However, even with all of this speculative computation going on, the retire unit forces the processor to appear as a simple RISC processor with predictable execution and interrupts.

Closing Notes

Congratulations for reaching the end of a long chapter! We have talked a little bit about old computers, CISC, RISC, post-RISC, and EPIC, and mentioned supercomputers in passing. I think it's interesting to observe that RISC processors are a branch off a long-established tree. Many of the ideas that have gone into RISC designs are borrowed from other types of computers, but none of them evolved into RISC — RISC started at a discontinuity. There were hints of a RISC revolution (the CDC 6600 and the IBM 801 project) but it really was forced on the world (for its own good) by CPU designers at Berkeley and Stanford in the 1980s.

As RISC has matured, there have been many improvements. Each time it appears that we have reached the limit of the performance of our microprocessors there is a new architectural breakthrough improving our single CPU performance. How long can it continue? It is clear that as long as competition continues, there is significant performance headroom using the out-of-order execution as the clock rates move from a typical 200 MHz to 500+ MHz. DEC's Alpha 21264 is planned to have four-way out-of-order execution at 500 MHz by 1998. As of 1998, vendors are beginning to reveal their plans for processors clocked at 1000 MHz or 1 GHz.

Unfortunately, developing a new processor is a very expensive task. If enough companies merge and competition diminishes, the rate of innovation will slow. Hopefully we will be seeing four processors on a chip, each 16-way out-of-order superscalar, clocked at 1 GHz for $200 before we eliminate competition and let the CPU designers rest on their laurels. At that point, scalable parallel processing will suddenly become interesting again.

How will designers tackle some of the fundamental architectural problems, perhaps the largest being memory systems? Even though the post-RISC architecture and the EPIC alleviate the latency problems somewhat, the memory bottleneck will always be there. The good news is that even though memory performance improves more slowly than CPU performance, memory system performance does improve over time. We'll look next at techniques for building memory systems.

As discussed in [link], the exercises that come at the end of most chapters in this book are not like the exercises in most engineering texts. These exercises are mostly thought experiments, without well-defined answers, designed to get you thinking about the hardware on your desk.

Exercises

Exercise:

Problem:

Speculative execution is safe for certain types of instructions; results can be discarded if it turns out that the instruction shouldn't have executed. Floating- point instructions and memory operations are two classes of instructions for which speculative execution is trickier, particularly because of the chance of generating exceptions. For instance, dividing by zero or taking the square root of a negative number causes an exception. Under what circumstances will a speculative memory reference cause an exception?

Exercise:

Problem:

Picture a machine with floating-point pipelines that are 100 stages deep (that's ridiculously deep), each of which can deliver a new result every nanosecond. That would give each pipeline a peak throughput rate of 1 Gflop, and a worst- case throughput rate of 10 Mflops. What characteristics would a program need to have to take advantage of such a pipeline?

Assembly Language

In this appendix, we take a look at the assembly language produced by a number of different compilers on a number of different architectures. In this survey we revisit some of the issues of CISC versus RISC, and the strengths and weaknesses of different architectures.

For this survey, two roughly identical segments of code were used. The code was a relatively long loop adding two arrays and storing the result in a third array. The loops were written both in FORTRAN and C.

The FORTRAN loop was as follows:

```
SUBROUTINE ADDEM(A,B,C,N)
REAL A(10000),B(10000),C(10000)
INTEGER N,I
DO 10 I=1,N
  A(I) = B(I) + C(I)
ENDDO
END
```

The C version was:

```
for(i=0;i<n;i++) a[i] = b[i] + c[i];
```

We have gathered these examples over the years from a number of different compilers, and the results are not particularly scientific. This is not intended to review a particular architecture or compiler version, but rather just to show an example of the kinds of things you can learn from looking at the output of the compiler.

Intel 8088

The Intel 8088 processor used in the original IBM Personal Computer is a very traditional CISC processing system with features severely limited by its transistor count. It has very few registers, and the registers generally have rather specific functions. To support a large memory model, it must set its segment register leading up to each memory operation. This limitation means that every memory access takes a minimum of three instructions. Interestingly, a similar pattern occurs on RISC processors.

You notice that at one point, the code moves a value from the `ax` register to the `bx` register because it needs to perform another computation that can only be done in the `ax` register. Note that this is only an integer computation, as the Intel

```
                mov     word ptr -2[bp],0           #
bp is I
      $11:
                mov     ax,word ptr -2[bp]          #
Load I
                cmp     ax,word ptr 18[bp]          #
Check I>=N
                bge     $10
                shl     ax,1                        #
Multiply I by 2
                mov     bx,ax                       #
Done - now move to bx
                add     bx,word ptr 10[bp]          #
bx = Address of B + Offset
                mov     es,word ptr 12[bp]          #
Top part of address
                mov     ax,es: word ptr [bx]        #
Load B(i)
                mov     bx,word ptr -2[bp]          #
Load I
                shl     bx,1                        #
Multiply I by 2
```

```
                add     bx,word ptr 14[bp]          #
bx = Address of C + Offset
                mov     es,word ptr 16[bp]          #
Top part of address
                add     ax,es: word ptr [bx]        #
Load C(I)
                mov     bx,word ptr -2[bp]          #
Load I
                shl     bx,1                        #
Multiply I by 2
                add     bx,word ptr 6[bp]           #
bx = Address of A + Offset
                mov     es,word ptr 8[bp]           #
Top part of address
                mov     es: word ptr [bx],ax        #
Store
      $9:
                inc     word ptr -2[bp]             #
Increment I in memory
                jmp $11
      $10:
```

Because there are so few registers, the variable `I` is kept in memory and loaded several times throughout the loop. The inc instruction at the end of the loop actually updates the value in memory. Interestingly, at the top of the loop, the value is then reloaded from memory.

In this type of architecture, the available registers put such a strain on the flexibility of the compiler, there is often not much optimization that is practical.

Motorola MC68020

In this section, we examine another classic CISC processor, the Motorola MC68020, which was used to build Macintosh computers and Sun

workstations. We happened to run this code on a BBN GP-1000 Butterfly parallel processing system made up of 96 MC68020 processors.

The Motorola architecture is relatively easy to program in assembly language. It has plenty of 32-bit registers, and they are relatively easy to use. It has a CISC instruction set that keeps assembly language programming quite simple. Many instructions can perform multiple operations in a single instruction.

We use this example to show a progression of optimization levels, using a f77 compiler on a floating-point version of the loop. Our first example is with no optimization:

```
                                                ! Note
d0 contains the value I
      L5:
            movl     d0,L13                     ! Store
I to memory if loop ends
            lea      a1@(-4),a0                 ! a1 =
address of B
            fmoves   a0@(0,d0:l:4),fp0          ! Load
of B(I)
            lea      a3@(-4),a0                 ! a3 =
address of C
            fadds    a0@(0,d0:l:4),fp0          ! Load
of C(I) (And Add)
            lea      a2@(-4),a0                 ! a2 =
address of A
            fmoves   fp0,a0@(0,d0:l:4)          ! Store
of A(I)
            addql    #1,d0                      !
Increment I
            subql    #1,d1                      !
Decrement "N"
            tstl     d1
```

```
bnes      L5
```

The value for `I` is stored in the `d0` register. Each time through the loop, it's incremented by `1`. At the same time, register `d1` is initialized to the value for `N` and decremented each time through the loop. Each time through the loop, `I` is stored into memory, so the proper value for `I` ends up in memory when the loop terminates. Registers `a1`, `a2`, and `a3` are preloaded to be the first address of the arrays `B`, `A`, and `C` respectively. However, since FORTRAN arrays begin at 1, we must subtract 4 from each of these addresses before we can use `I` as the offset. The `lea` instructions are effectively subtracting 4 from one address register and storing it in another.

The following instruction performs an address computation that is almost a one-to- one translation of an array reference:

```
fmoves a0@(0,d0:l:4),fp0 ! Load of B(I)
```

This instruction retrieves a floating-point value from the memory. The address is computed by first multiplying `d0` by 4 (because these are 32-bit floating-point numbers) and adding that value to `a0`. As a matter of fact, the `lea` and `fmoves` instructions could have been combined as follows:

```
fmoves a1@(-4,d0:l:4),fp0 ! Load of B(I)
```

To compute its memory address, this instruction multiplies `d0` by 4, adds the contents of `a1`, and then subtracts 4. The resulting address is used to load 4 bytes into floating-point register `fp0`. This is almost a literal translation of fetching `B(I)`. You can see how the assembly is set up to track high-level constructs.

It is almost as if the compiler were “trying” to show off and make use of the nifty assembly language instructions.

Like the Intel, this is not a load-store architecture. The `fadds` instruction adds a value from memory to a value in a register (`fp0`) and leaves the result of the addition in the register. Unlike the Intel 8088, we have enough registers to store quite a few of the values used throughout the loop (`I`, `N`, the address of `A`, `B`, and `C`) in registers to save memory operations.

C on the MC68020

In the next example, we compiled the C version of the loop with the normal optimization (`-O`) turned on. We see the C perspective on arrays in this code. C views arrays as extensions to pointers in C; the loop index advances as an offset from a pointer to the beginning of the array:

```
      ! d3 = I
      ! d1 = Address of A
      ! d2 = Address of B
      ! d0 = Address of C
      ! a6@(20) = N
              moveq   #0,d3                   !
Initialize I
              bras    L5                      ! Jump
to End of the loop
      L1:     movl    d3,a1                   ! Make
copy of I
              movl    a1,d4                   !
Again
              asll    #2,d4                   !
Multiply by 4 (word size)
              movl    d4,a1                   ! Put
back in an address register
              fmoves  a1@(0,d2:l),fp0         ! Load
```

```
B(I)
                movl    a6@(16),d0              ! Get
address of C
                fadds   a1@(0,d0:l),fp0         ! Add
C(I)
                fmoves  fp0,a1@(0,d1:l)         !
Store into A(I)
                addql   #1,d3                   !
Increment I
      L5:
                cmpl    a6@(20),d3
                bits    L1
```

We first see the value of `I` being copied into several registers and multiplied by 4 (using a left shift of 2, strength reduction). Interestingly, the value in register `a1` is `I` multiplied by 4. Registers `d0`, `d1`, and `d2` are the addresses of `C`, `B`, and `A` respectively. In the load, add, and store, `a1` is the base of the address computation and `d0`, `d1`, and `d2` are added as an offset to `a1` to compute each address.

This is a simplistic optimization that is primarily trying to maximize the values that are kept in registers during loop execution. Overall, it's a relatively literal translation of the C language semantics from C to assembly. In many ways, C was designed to generate relatively efficient code without requiring a highly sophisticated optimizer.

More optimization

In this example, we are back to the FORTRAN version on the MC68020. We have compiled it with the highest level of optimization (`-OLM`) available on this compiler. Now we see a much more aggressive approach to the loop:

```
        ! a0 = Address of C(I)
```

```
      ! a1 = Address of B(I)
      ! a2 = Address of A(I)
      L3:
              fmoves  a1@,fp0                    !
Load B(I)
              fadds   a0@,fp0                    !
Add C(I)
              fmoves  fp0,a2@                    !
Store A(I)
              addql   #4,a0                      !
Advance by 4
              addql   #4,a1                      !
Advance by 4
              addql   #4,a2                      !
Advance by 4
              subql   #1,d0                      !
Decrement I
              tstl    d0
              bnes    L3
```

First off, the compiler is smart enough to do all of its address adjustment outside the loop and store the adjusted addresses of `A`, `B`, and `C` in registers. We do the load, add, and store in quick succession. Then we advance the array addresses by 4 and perform the subtraction to determine when the loop is complete.

This is very tight code and bears little resemblance to the original FORTRAN code.

SPARC Architecture

These next examples were performed using a SPARC architecture system using FORTRAN. The SPARC architecture is a classic RISC processor using load-store access to memory, many registers and delayed branching. We first examine the code at the lowest optimization:

```
      .L18:                                     ! Top
of the loop
              ld      [%fp-4],%l2               !
Address of B
              sethi   %hi(GPB.addem.i),%l0      !
Address of I in %l0
              or      %l0,%lo(GPB.addem.i),%l0
              ld      [%l0+0],%l0               ! Load
I
              sll     %l0,2,%l1                 !
Multiply by 4
              add     %l2,%l1,%l0               !
Figure effective address of B(I)
              ld      [%l0+0],%f3               ! Load
B(I)
              ld      [%fp-8],%l2               !
Address of C
              sethi   %hi(GPB.addem.i),%l0      !
Address of I in %l0
              or      %l0,%lo(GPB.addem.i),%l0
              ld      [%l0+0],%l0               ! Load
I
              sll     %l0,2,%l1                 !
Multiply by 4
              add     %l2,%l1,%l0               !
Figure effective address of B(I)
              ld      [%l0+0],%f2               ! Load
C(I)
              fadds   %f3,%f2,%f2               ! Do
the Floating Point Add
              ld      [%fp-12],%l2              !
Address of A
              sethi   %hi(GPB.addem.i),%l0      !
Address of i in %l0
              or      %l0,%lo(GPB.addem.i),%l0
```

```
              ld      [%l0+0],%l0           ! Load
I
              sll     %l0,2,%l1             !
Multiply by 4
              add     %l2,%l1,%l0           !
Figure effective address of A(I)
              st      %f2,[%l0+0]           !
Store A(I)
              sethi   %hi(GPB.addem.i),%l0  !
Address of i in %l0
              or      %l0,%lo(GPB.addem.i),%l0
              ld      [%l0+0],%l0           ! Load
I
              add     %l0,1,%l1             !
Increment I
              sethi   %hi(GPB.addem.i),%l0  !
Address of I in %l0
              or      %l0,%lo(GPB.addem.i),%l0
              st      %l1,[%l0+0]           !
Store I
              sethi   %hi(GPB.addem.i),%l0  !
Address of I in %l0
              or      %l0,%lo(GPB.addem.i),%l0
              ld      [%l0+0],%l1           ! Load
I
              ld      [%fp-20],%l0          ! Load
N
              cmp     %l1,%l0               !
Compare
              ble     .L18
              nop                           !
Branch Delay Slot
```

This is some pretty poor code. We don't need to go through it line by line, but there are a few quick observations we can make. The value for `I` is loaded from memory five times in the loop. The address of `I` is computed

six times throughout the loop (each time takes two instructions). There are no tricky memory addressing modes, so multiplying `I` by 4 to get a byte offset is done explicitly three times (at least they use a shift). To add insult to injury, they even put a NO-OP in the branch delay slot.

One might ask, “Why do they ever generate code this bad?” Well, it’s not because the compiler isn’t capable of generating efficient code, as we shall see below. One explanation is that in this optimization level, it simply does a one-to-one translation of the tuples (intermediate code) into machine language. You can almost draw lines in the above example and precisely identify which instructions came from which tuples.

One reason to generate the code using this simplistic approach is to guarantee that the program will produce the correct results. Looking at the above code, it’s pretty easy to argue that it indeed does exactly what the FORTRAN code does. You can track every single assembly statement directly back to part of a FORTRAN statement.

It’s pretty clear that you don’t want to execute this code in a high performance production environment without some more optimization.

Moderate optimization

In this example, we enable some optimization (`-O1`):

```
            save    %sp,-120,%sp            !
Rotate the register window
            add     %i0,-4,%o0              !
Address of A(0)
            st      %o0,[%fp-12]            ! Store
on the stack
            add     %i1,-4,%o0              !
Address of B(0)
            st      %o0,[%fp-4]             ! Store
```

```
on the stack
            add     %i2,-4,%o0              !
Address of C(0)
            st      %o0,[%fp-8]             ! Store
on the stack
            sethi   %hi(GPB.addem.i),%o0    !
Address of I (top portion)
            add     %o0,%lo(GPB.addem.i),%o2 !
Address of I (lower portion)
            ld      [%i3],%o0               ! %o0 =
N (fourth parameter)
            or      %g0,1,%o1               ! %o1 =
1 (for addition)
            st      %o0,[%fp-20]            ! store
N on the stack
            st      %o1,[%o2]               ! Set
memory copy of I to 1
            ld      [%o2],%o1               ! o1 =
I (kind of redundant)
            cmp     %o1,%o0                 ! Check
I > N (zero-trip?)
            bg      .L12                    ! Don't
do loop at all
            nop                              !
Delay Slot
            ld      [%o2],%o0               ! Pre-
load for Branch Delay Slot
      .L900000110:                           ! Top
of the loop
            ld      [%fp-4],%o1             ! o1 =
Address of B(0)
            sll     %o0,2,%o0               !
Multiply I by 4
            ld      [%o1+%o0],%f2           ! f2 =
B(I)
            ld      [%o2],%o0               ! Load
I from memory
```

```
            ld      [%fp-8],%o1             ! o1 =
Address of C(0)
            sll     %o0,2,%o0               !
Multiply I by 4
            ld      [%o1+%o0],%f3           ! f3 =
C(I)
            fadds   %f2,%f3,%f2             !
Register-to-register add
            ld      [%o2],%o0               ! Load
I from memory (not again!)
            ld      [%fp-12],%o1            ! o1 =
Address of A(0)
            sll     %o0,2,%o0               !
Multiply I by 4 (yes, again)
            st      %f2,[%o1+%o0]           ! A(I)
= f2
            ld      [%o2],%o0               ! Load
I from memory
            add     %o0,1,%o0               !
Increment I in register
            st      %o0,[%o2]               ! Store
I back into memory
            ld      [%o2],%o0               ! Load
I back into a register
            ld      [%fp-20],%o1            ! Load
N into a register
            cmp     %o0,%o1                 ! I > N
??
            ble,a   .L900000110
             ld      [%o2],%o0               !
Branch Delay Slot
```

This is a significant improvement from the previous example. Some loop constant computations (subtracting 4) were hoisted out of the loop. We only loaded `I` 4 times during a loop iteration. Strangely, the compiler didn't choose to store the addresses of `A(0)`, `B(0)`, and `C(0)` in registers at all

even though there were plenty of registers. Even more perplexing is the fact that it loaded a value from memory immediately after it had stored it from the exact same register!

But one bright spot is the branch delay slot. For the first iteration, the load was done before the loop started. For the successive iterations, the first load was done in the branch delay slot at the bottom of the loop.

Comparing this code to the moderate optimization code on the MC68020, you can begin to get a sense of why RISC was not an overnight sensation. It turned out that an unsophisticated compiler could generate much tighter code for a CISC processor than a RISC processor. RISC processors are always executing extra instructions here and there to compensate for the lack of slick features in their instruction set. If a processor has a faster clock rate but has to execute more instructions, it does not always have better performance than a slower, more efficient processor.

But as we shall soon see, this CISC advantage is about to evaporate in this particular example.

Higher optimization

We now increase the optimization to `-O2`. Now the compiler generates much better code. It’s important you remember that this is the same compiler being used for all three examples.

At this optimization level, the compiler looked through the code sufficiently well to know it didn’t even need to rotate the register windows (no save instruction). Clearly the compiler looked at the register usage of the entire routine:

```
      ! Note, didn’t even rotate the register
Window
      ! We just use the %o registers from the
```

caller

```
      ! %o0 = Address of first element of A (from
calling convention)
      ! %o1 = Address of first element of B (from
calling convention)
      ! %o2 = Address of first element of C (from
calling convention)
      ! %o3 = Address of N (from calling
convention)

      addem_:
             ld      [%o3],%g2                 !
Load N
             cmp     %g2,1                     !
Check to see if it is <1
             bl      .L77000006                !
Check for zero trip loop
             or      %g0,1,%g1                 !
Delay slot - Set I to 1
       .L77000003:
             ld      [%o1],%f0                 !
Load B(I) First time Only
       .L900000109:
             ld      [%o2],%f1                 !
Load C(I)
             fadds   %f0,%f1,%f0               !
Add
             add     %g1,1,%g1                 !
Increment I
             add     %o1,4,%o1                 !
Increment Address of B
             add     %o2,4,%o2                 !
Increment Address of C
             cmp     %g1,%g2                   !
Check Loop Termination
             st      %f0,[%o0]                 !
```

```
Store A(I)
            add     %o0,4,%o0                   !
Increment Address of A
            ble,a   .L900000109                 !
Branch w/ annul
            ld      [%o1],%f0                   !
Load the B(I)
      .L77000006:
            retl                                !
Leaf Return (No window)
            nop                                 !
Branch Delay Slot
```

This is tight code. The registers `o0`, `o1`, and `o2` contain the addresses of the first elements of `A`, `B`, and `C` respectively. They already point to the right value for the first iteration of the loop. The value for `I` is never stored in memory; it is kept in global register `g1`. Instead of multiplying `I` by 4, we simply advance the three addresses by 4 bytes each iteration.

The branch delay slots are utilized for both branches. The branch at the bottom of the loop uses the `annul` feature to cancel the following load if the branch falls through.

The most interesting observation regarding this code is the striking similarity to the code and the code generated for the MC68020 at its top optimization level:

```
L3:
              fmoves  a1@,fp0                    !
Load B(I)
              fadds   a0@,fp0                   !
Add C(I)
              fmoves  fp0,a2@                   !
Store A(I)
              addql   #4,a0                     !
Advance by 4
```

```
                addql   #4,a1                        !
Advance by 4
                addql   #4,a2                        !
Advance by 4
                subql   #1,d0                        !
Decrement I
                tstl    d0
                bnes    L3
```

The two code sequences are nearly identical! For the SPARC, it does an extra load because of its load-store architecture. On the SPARC, `I` is incremented and compared to `N`, while on the MC68020, `I` is decremented and compared to zero.

This aptly shows how the advancing compiler optimization capabilities quickly made the “nifty” features of the CISC architectures rather useless. Even on the CISC processor, the post-optimization code used the simple forms of the instructions because they produce they fastest execution time.

Note that these code sequences were generated on an MC68020. An MC68060 should be able to eliminate the three `addql` instructions by using post-increment, saving three instructions. Add a little loop unrolling, and you have some very tight code. Of course, the MC68060 was never a broadly deployed workstation processor, so we never really got a chance to take it for a test drive.

Convex C-240

This section shows the results of compiling on the Convex C-Series of parallel/vector supercomputers. In addition to their normal registers, vector computers have vector registers that contain up to 256 64-bit elements. These processors can perform operations on any subset of these registers with a single instruction.

It is hard to claim that these vector supercomputers are more RISC or CISC. They have simple lean instruction sets and, hence, are RISC-like. However,

they have instructions that implement loops, and so they are somewhat CISC-like.

The Convex C-240 has scalar registers `(s2)`, vector registers `(v2)`, and address registers `(a3)`. Each vector register has 128 elements. The vector length register controls how many of the elements of each vector register are processed by vector instructions. If vector length is above 128, the entire register is processed.

The code to implement our loop is as follows:

```
      L4:       mov.ws   2,vl
; Set the Vector length to N
                ld.w     0(a5),v0
; Load B into Vector Register
                ld.w     0(a2),v1
; Load C into Vector Register
                add.s    v1,v0,v2
; Add the vector registers
                st.w     v2,0(a3)
; Store results into A
                add.w    #-128,s2
; Decrement "N"
                add.w    #512,a2
; Advance address for A
                add.w    #512,a3
; Advance address for B
                add.w    #512,a5
; Advance address for C
                lt.w     #0,s2
; Check to see if "N" is < 0
                jbrs.t   L4
```

Initially, the vector length register is set to `N`. We assume that for the first iteration, `N` is greater than 128. The next instruction is a vector load instruction into register v0. This loads 128 32-bit elements into this register. The next instruction also loads 128 elements, and the following instruction adds those two registers and places the results into a third vector register. Then the 128 elements in Register v2 are stored back into memory. After those elements have been processed, `N` is decremented by 128 (after all, we did process 128 elements). Then we add 512 to each of the addresses (4 bytes per element) and loop back up. At some point, during the last iteration, if `N` is not an exact multiple of 128, the vector length register is less than 128, and the vector instructions only process those remaining elements up to `N`.

One of the challenges of vector processors is to allow an instruction to begin executing before the previous instruction has completed. For example, once the load into Register v1 has partially completed, the processor could actually begin adding the first few elements of v0 and v1 while waiting for the rest of the elements of v1 to arrive. This approach of starting the next vector instruction before the previous vector instruction has completed is called chaining. Chaining is an important feature to get maximum performance from vector processors.

IBM RS-6000

The IBM RS-6000 is generally credited as the first RISC processor to have cracked the Linpack 100×100 benchmark. The RS-6000 is characterized by strong floating-point performance and excellent memory bandwidth among RISC workstations. The RS-6000 was the basis for IBM’s scalable parallel processor: the IBM-SP1 and SP2.

When our example program is run on the RS-6000, we can see the use of a CISC- style instruction in the middle of a RISC processor. The RS-6000 supports a branch- on-count instruction that combines the decrement, test, and branch operations into a single instruction. Moreover, there is a special register (the count register) that is part of the instruction fetch unit that stores the current value of the counter. The fetch unit also has its own add unit to perform the decrements for this instruction.

These types of features creeping into RISC architectures are occuring because there is plenty of chip space for them. If a wide range of programs can run faster with this type of instruction, it's often added.

The assembly code on the RS-6000 is:

```
            ai       r3,r3,-4                 #
Address of A(0)
            ai       r5,r5,-4                 #
Address of B(0)
            ai       r4,r4,-4                 #
Address of C(0)
            bcr      BO_IF_NOT,CR0_GT
            mtspr    CTR,r6                   #
Store in the Counter Register
      __L18:
            lfsu     fp0,4(r4)                #
Pre Increment Load
            lfsu     fp1,4(r5)                #
Pre Increment Load
            fa       fp0,fp0,fp1
            frsp     fp0,fp0
            stfsu    fp0,4(r3)                #
Pre-increment Store
            bc       BO_dCTR_NZERO,CR0_LT,__L18
# Branch on Counter
```

The RS-6000 also supports a memory addressing mode that can add a value to its address register before using the address register. Interestingly, these two features (branch on count and pre-increment load) eliminate several instructions when compared to the more "pure" SPARC processor. The SPARC processor has 10 instructions in the body of its loop, while the RS-6000 has 6 instructions.

The advantage of the RS-6000 in this particular loop may be less significant if both processors were two-way superscalar. The instructions were eliminated on the RS-6000 were integer instructions. On a two-way superscalar processor, those integer instructions may simply execute on the integer units while the floating-point units are busy performing the floating-point computations.

Conclusion

In this section, we have attempted to give you some understanding of the variety of assembly language that is produced by compilers at different optimization levels and on different computer architectures. At some point during the tuning of your code, it can be quite instructive to take a look at the generated assembly language to be sure that the compiler is not doing something really stupid that is slowing you down.

Please don’t be tempted to rewrite portions in assembly language. Usually any problems can be solved by cleaning up and streamlining your high-level source code and setting the proper compiler flags.

It is interesting that very few people actually learn assembly language any more. Most folks find that the compiler is the best teacher of assembly language. By adding the appropriate option (often `-S`), the compiler starts giving you lessons. I suggest that you don’t print out all of the code. There are many pages of useless variable declarations, etc. For these examples, I cut out all of that useless information. It is best to view the assembly in an editor and only print out the portion that pertains to the particular loop you are tuning.